

LIBRARY OF PETTM SUBROUTINES

NICK HAMPSHIRE

A NICK HAMPSHIRE COMMODORE PUBLICATION

First Edition June 1980

The programs presented in this book have been included for their instructional value, they have been checked out with care, however, they are not warranted for any purpose. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for any errors or omissions. Neither is any liability assumed for damages or other costs resulting from the use of the information contained herein. No patent liability is assumed for the information contained herein nor do the publishers assume any liability for infringement of patents or other rights of third parties resulting from use of that information. No licence is granted by the equipment manufacturers under any patent or patent rights and manufacturers reserve the right to change circuitry and software at any time without notice. Readers are referred to current manufacturers data for exact specifications.

COPYRIGHT 1980 Nick Hampshire. World rights reserved. No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including, but not limited to, photocopy, photography, magnetic or other recording, without prior written permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

Published by Commodore Business Machines (UK) Limited,
818 Leigh Road, Trading Estate, Slough, Berkshire.
Telephone: Slough (0753) 74111 Telex: 848403
PET is a Trademark of Commodore Ltd.

INTRODUCTION

For the average PET user the prospect of writing a set of programs to perform, say, a business application is daunting. Many don't even try, preferring to employ someone to do the job or buy an off-the-shelf package. What is daunting is not simply writing a program, every PET user will probably have written dozens of simple little programs for his own use. It is the size of a set of application programs, the thought of writing several thousand lines of Basic code. Allied to this is the question of how and where to start, and what exactly one wants the program to do. These problems are purely a mental block, since any person of average intelligence can write and design such a program. Given a logical framework on which to build, plus a few aids in the form of standard subroutines, the process of writing an applications program becomes considerably easier.

Defining the Program Structure.

The process of writing and running a program is one of defining a problem or process from the very general level of thought used by the human mind to the very precise sequence of instructions used by the machine. The lowest, and therefore the most precise level, is the binary code used by the microprocessor; this is the level used in machine code programs. At a higher level are the commands of a Basic interpreter. These allow instructions to be given to the processor in easily understood English language statements. The execution of one instruction statement in Basic may require several hundred steps of machine code program. The next level above the Basic interpreter is the subroutine. This is a block of code written in Basic to perform a specific function. Subroutines give the programmer higher level functions than those available in Basic. Thus, a subroutine to sort an array into alphanumeric order gives the programmer a new command, summarised as: SORT ARRAY X. Just as the Basic command represented several hundred machine code commands, so a subroutine may consist of a hundred Basic commands. The first level of command generalisation above the subroutine is the program, consisting of a collection of

subroutines. Above the program is the highest level, the suite of programs, this is a collection of several programs which are united by a common data base and collectively perform an application, eg: stock control.

The existence of a hierarchy of conceptual generalisations is very useful since it allows the process of program writing to be split into a series of easily defined stages. This firstly gives the programmer a logical sequence of operations and secondly allows much of the work associated with the lowest levels to be automated. This is done by using a high level language like Basic which removes the necessity of writing in the lowest or machine code level. Likewise the amount of Basic code required to be written can be considerably reduced by using standard subroutines. Entire programs within a program suite may even be standard and therefore usable in many different program suites.

The first stage in writing a large applications program is to define the exact function of the program suite. It is also important at this stage to determine what hardware will be used by the program, together with the limitations and capabilities of that hardware. It is necessary to know what hardware is being used since this determines the size of the program, and the nature and organisation of data files used by the program. Thus the approach taken in writing a program for a 32K PET with 3040 disk and 3022 printer will be totally different to that taken for the same application on an 8K PET with cassette deck. With these limitations in mind, the first stage is to write a short description of the application, with a definition of the input, output, and data files.

Consider a hypothetical user with a 32K PET, dual 2040 floppy disks and 3022 printer wishing to use this system to implement a library reference data base. The first stage in system design is for the user to decide exactly what the program must do. In this case the program is required to reference a book or books from a small library of 500 titles by either subject matter or author. The user wants to type in a subject in which he is interested and the computer to produce a list of book titles in his library containing information on that subject. Similarly the user wants to produce a list of books written by a particular author. The problem is a fairly straight forward one of data access and retrieval from a data base stored on disk. The program can be divided into four parts:

- 1)Data entry: used to enter details of new titles added to the library.
- 2)Data update: needed to correct mistakes in the data base or to delete entries whose titles have been removed from the library.
- 3)Data access: the part of the program which performs the user's requirement of accessing data from the data base in response to a particular input.

4)Data file maintenance: to allow the user to make security backup copies of the data file. It will also perform functions such as sorting the data file into alphabetical order.

You will notice that of the four parts of this program three parts are concerned with the upkeep of the data file. This is typical of all programs using disk data bases. Each of these four program parts are totally independent of the other parts and interact with each other only via the data base. Since each part is independent it can be written as a separate program, which makes life much simpler for the programmer - only one part of the program need be written at a time. Each part can be stored on disk as a separate program and loaded by the user when required. A collection of programs like this constitutes the program suite.

File Structure.

The factor unifying all the programs in a suite is that they all use the same data base. Before any programs are written, the nature and format of the data must be defined. This is probably the most critical and trickiest part in the design of a program: bad file structures are the cause of a lot of poor programs. Unfortunately there is no easy rule of thumb which can be used to select the best kind of data files to be used in a program. The only rule worth remembering is that if the data file is very large, by which I mean a file which contains more data than can be stored in the machine's internal memory, then that file should in the majority of cases be a random access file. Short files which can be loaded into arrays and stored in core are best stored on disk as sequential files. The reason for this rule is that it makes data access times considerably faster and also makes file maintenance much easier.

There are three principal ways in which data can be stored: sequential disk (or tape)files, direct access disk files, and data stored within a program as data statements. Each of these methods has its advantages and disadvantages and the selection of method used depends on finding which is best suited to the application. Sequential files are the simplest to use since the operating system automatically organises the information. In a sequential file, data items are recorded one after the other to create a chain of data, with the first item recorded at the head and the last item at the tail of the chain, in "sequential" order. The virtue of sequential files is that data items can be of variable length, thereby eliminating wasted disk or tape storage space. The disadvantage of sequential files is that each time a record is read the entire file in front of that record must be read, since there is no way in which the program can determine the starting location of a particular record. Also,

the only way in which an item can be inserted or amended in a sequential file is by writing an updated version of the entire file.

Direct access files differ from sequential files in that one can specify the track, the sector on that track, and the character within that sector on which a record is to be recorded. Provided a record of the location of each record is kept, then direct access files allow one to go straight to a record without reading all preceding records on the file. Direct access files thus allow information to be accessed in a non-sequential or random order (for this reason direct access files are usually known as random access files). It is unnecessary to rewrite the entire file to amend a record on a direct access file. However, records on a random access file must have a fixed length, otherwise the task of keeping track of the location of each record becomes excessively complex. The use of fixed length records means that direct access files use more space than sequential access files to store the same data. The primary advantage of random access files is speed of access. The disadvantage of direct access on the PET depends on which version of Basic and disk operating system is installed in your machine. Machines using Basic 3.00 and DOS 1.00 require fairly complex programming to create and use direct access files. This has been cured on later machines using Basic 4.00 and DOS 2.00. These machines have simple to use relative record file commands. Relative record means that data is accessed by record number, ie. record 50 is the 50th record on the file. Direct access is the principal choice for all data base files where large quantities of data are to be stored and accessed by a program.

It is desirable in some cases to store data directly within a program as data statements. This has the advantage of rapid access and is particularly useful in tape based PET systems. Data can be added or amended using an automatic data statement generator subroutine. Data statements can be accessed either by a sequential search or by a relative record method. The obvious disadvantage of storing data within a program as data statements is that since all data is stored in core the maximum size of the data file is limited to the size of available memory. Another disadvantage which is shared by sequential files stored in arrays is that data entered or updated can be lost if the machine is switched off prior to saving the new program or data on tape or disk.

In the example program, careful examination of the problem reveals that three major data files are required. The largest is the primary data base file which contains book titles and details of the contents: this file is to be organised as a random access file. The other two files are both sequential: an author file and a subject file. It is one of these two shorter files which is searched during data access. Each record contains, say, the author's name followed

by a series of pointers to records in the random access main data base file. These pointers are simply record numbers: in a random access file system this is all the information required to access a particular block of data from a file. The advantage of using a short file consisting of a key and one or more pointers into a larger file is primarily one of speed, but also it is much easier to sort a short key file than a long random access file. Another advantage in using short key files is that one can have multiple key files, just as here we have an author file and a subject file.

Having decided the types of file to be used in the program suite, the example is very typical of many applications programs, we must decide on what data format is to be used in those files. Very careful thought should be given to data formatting if optimum usage is to be made of the storage capacity of the machine. First let's look at random access files. A random access file consists of a set of records: there may be one hundred records, a thousand or even ten thousand, the number depending on the application. In the example the maximum number of titles is 500 therefore we do not require more than 500 records in the random file. Each record is of the same length, common record lengths are 128 or 256 bytes or some other power of 2, since using a record 256 bytes long makes for greater reliability because this is the natural organisation of the computer system. Since the PET disk is divided into 256 byte blocks it is easiest if we use one block per random access record, giving a practical maximum of 640 records on a disk. The first step is to jot down a list of all the different data you wish stored in each record and against each item to note the maximum number of bytes required to store it, together with a note of whether the data is numeric or alpha. A running total of the number of bytes used should be kept since it is very important that the total should not exceed the maximum which we have set for the record, in this case 256 bytes. Also great care should be taken that each data item, or field as it is known, within a record has sufficient space. The space must accommodate the maximum possible size data entry necessary, to ensure transfer of all required information in that field. In our example the list would probably look something like this:

AUTHOR	30	Alpha
TITLE	60	Alpha
DATE	6	Numeric
CONTENTS1	80	Alpha
CONTENTS2	80	Alpha

There are four alpha fields in the record and one numeric. Records should be stored so that all the fields are of the same variable type. In the example the date should be converted to an alpha string before it is entered into the

record. The date field always occupies 6 bytes but the four alpha fields are of variable length; the number of bytes quoted in Table 1 are the maximum for each entry. Any entry shorter than the maximum is padded out with spaces on the end of the string to bring it up to the maximum length. For instance Shakespeare, W. will be stored as SHAKESPEARE, W.----- (the dashes represent spaces). Using this method we know that the title field always starts at the 31st byte of a record and ends on the 90th byte, making it easier to use the string manipulation commands in Basic to retrieve each item of data from a record.

0	20	70	76	255
Author	Title	Date	Contents	

Since it is envisaged that our example random access file will have a maximum of 500 records each of 256 bytes, the total data base will have a maximum size of 125K bytes. This will fit nicely on one disk, a fact which should always be aimed for when designing a program since it is good practice to use one disk drive for data and the other for program storage. By the same token it is not desirable to change disks during program execution since this invariably leads to unreliability and a greatly increased chance of losing a data file.

While random access records must always have a fixed length, records in sequential files can have either fixed or variable lengths. The choice depends on the application. In many cases the data will always be the same length, making a fixed length and format sequential file the logical choice. Or as with the random access file in our example each sequential file record may contain multiple fields, so fixed format, fixed length records are the answer. They are much easier to dissect than variable length records. But there are applications, and our example is one of them, where fixed length sequential records are impractical. Each of the two sequential files contains a variable number of records. Each record consists of a key (the author's name or his subject) and a variable number of numeric pointers to records in the random access file. To use fixed length records in this application would require the allocation of sufficient space in each record to accommodate the maximum length key word together with the maximum number of pointers. Since the file is to be stored in core memory as an array, the large amount of space (and therefore memory) wasted by using fixed length records makes this totally impractical. When records do not have fixed lengths, demarcation markers must be used between fields. The marker most commonly used is an asterisk. To dissect this type of record, search from left to right for the first occurrence of the marker character. The record to

the right of the marker is removed and the next marker searched for. This method is clumsier than that used for fixed length fixed format records, but is just as effective. In our example a record in the author file may be stored as:

SMITH.J*14*56*79*125*256*428*0

The first field is a key - the author's name - SMITH.J. The next and subsequent fields are the pointers to the random access file showing that titles written by this author are stored in records 14,56,79,125,256,and 428. The zero is used as an end of record marker. Using this method the author's name can be as long as necessary and can be followed by as many references as are required (subject only to the maximum length of a record which is limited by the 80 character buffer).

Whatever file type is used, pointers must be used to locate the last entry on the file. This is vital if one is to be able to add further data to the file without risk of erasing existing data. In a random access file record zero could be allocated for this purpose, providing that the programmer remembers that this record must only be used to contain the number of the next free record. Alternatively this information could be stored, together with other information essential for the correct operation of the system, in a special parameter file. Parameter files are very useful in any program suite, since they not only allow variables to be passed between programs or stored between runs, but, they also allow a package to be amended to suit a particular user's requirements. In a sequential access file the easiest method is to use a special end record as the last record on the file. This way one simply continues to read records from the file until the end record is encountered. The usual form of end record is one filled with a string of Z's thus:

ZZ

It is then a simple matter to check if, say, the first three bytes of a record are "ZZZ", since the odds against this combination occurring naturally in text are large it is a reliable indication of the end of the file. The actual number of records in a sequential file can be obtained, as the file is read, by counting the records. Since the file will be stored in core, records can be easily added and inserted in the correct location using a sort procedure. The new file created in this way is then rewritten over the old file. If the file becomes larger than the available array area it will have to be broken into two sub files, as we have done with the author and subject files which could have been merged as one file if core space permitted.

The Program Specifications.

In these first stages we have defined how the data will be stored by the program and what functions it is desired that the program perform. From these decisions a program specification can be written. This is the framework of the program around which the actual coding will be built. It is important that it is correct in every detail since a mistake at this stage may prove very difficult to correct later on. These specifications also form the base around which the final user documentation can be written. The specification should contain a complete description of the operation of each program in the suite showing what data is required from the operator, what data is to be output and which data files are accessed. This written description should be accompanied by a diagram showing the flow of data. The second part of the specifications should be a complete table of the data file structures used by the suite. The third and last part should be a description of how data input, and output (either on the screen or on a printer), should be formatted. This may seem to be just a cosmetic operation, but the ease with which an operator can use the system is almost entirely dependent on the thought put into this part of the specification.

At each stage in the program the screen layout and/or printer output should be drawn out on a piece of squared paper. A table should be made of standard forms of input to be used throughout the programs. One example would be the format of all date inputs. All inputs ought to be made using the same variable type and format since it is disturbing for the operator to have some inputs terminated by a carriage return and others not. One might make a decision to have all inputs in string format with a fixed length prompt and traps for null entries, illegal carriage returns, etc. Decisions should also be made at this stage as to the nature of any input validation and error checking procedures, such as the use of a check digit.

It is a good idea to use flow charts at every stage in the design of a computer program. By displaying a process in a visual form it is usually much easier to detect faults in logic flow. Flow charts of program sequence and logic should be drawn for each level of the hierarchy of program definition. As the definition becomes more precise, process steps on the general definition flowchart can be replaced by flow chart modules more accurately defining that process step. Process steps corresponding to standard subroutines need not of course be defined as precisely as the rest of the program. Extensive use of standard subroutines helps reduce the complexity of the final flowchart. Each step on the final flow chart will correspond either to a subroutine or one or more Basic commands. This final version of the flow chart can be used as a guide when writing the program code.

Subroutines.

Once the specifications have been written, programming can begin. When working on a suite of programs it is best to start with data entry, since this program can then be used to create the data files needed to test the other programs in the suite. Each of these programs can be divided into a set of subroutines. A subroutine is a short, self contained program written to perform a specific function, which can be run by itself without the rest of the main program. The great virtue of using standard subroutines to build a program is that they need not be specific to that program, and can be reused in any other program requiring them. This means that once a good subroutine has been written it can be repeatedly used in other programs greatly reducing programming effort. The extensive use of standard subroutines is the key to writing good programs quickly and easily.

Parameters.

To make maximum use of standard subroutines the programmer must adopt a strict discipline about the use of variable names. This is essential since a subroutine communicates with the main body of the program (ie the code which links the subroutines together) by means of input and output variables. Obviously a specific set of variables must be reserved for this purpose. These variables are known as parameters and there are several ways in which they can be supplied, or passed, between a subroutine and the main program or another subroutine. In this book we will use two methods of passing parameters. The first is known as "call by value and result" and the second as "call by reference". Formal methods of passing parameters are needed because the variable names used in a subroutine are unlikely to be the same as those used in the main program. Thus a variable used within a subroutine may be called PX but the value of PX is obtained from the main program. This subroutine may be called several times in a program, but the value to be passed to PX (this is called a formal parameter) might be stored in the first case in variable A and in the second case in variable Q (both A and Q are actual parameters), and so on. Unless a value is placed in the formal parameter from the main program, by setting it equal to the actual parameter used in the main program, then the result of jumping to the subroutine will be wrong.

Call by value and result is the commonest and most straightforward method of passing parameters between subroutines written in Basic. This requires two parameter passing operations - one before jumping to the subroutine and the other after returning from the subroutine. In the first operation the values of the actual parameters are moved to the formal parameters, these are the parameters used by the

subroutine. In the second operation the values of the parameters set by the subroutine are moved from the formal to the actual parameters. As an example: we wish to call a subroutine to perform an arithmetic operation on two variables and produce the result as a third variable. The two actual parameters used by the subroutine are stored in this case in variables A and B, and the actual result parameter in variable C. The formal parameters used by the subroutine are variables PX and PY, the formal parameter set by the subroutine is variable PZ.

Operation 1 - PX=A:PY=B

Jump to subroutine.

Operation 2 - C=PZ

In this particular case all the formal parameters are either used or set, however, this may not always be the case. Suppose in the above example the subroutine has a fourth formal parameter PQ which is only set to one if the result of the arithmetic operation is negative. Sometimes the value of PQ will be 1 and for the remainder of the time 0. However since only a negative result is being tested for, then once PQ is set to 1 it will remain at 1 for all subsequent calls of that subroutine (unless set to 0 prior to calling the subroutine). In another situation only the formal parameter PX may be passed to the subroutine instead of PX and PY. Here one would normally want the value of PY to default to zero, unless PY is actually set to 0 the value will be that entered when the subroutine was last called. In these cases we need to know which parameters are used and which are set. The standard method is to regard all parameters as both used and returned at all times, meaning that where no actual parameters are passed to formal parameters (as in the case of parameters used only for results) the formal parameters should be set to their default value. The two operations for passing parameters now become:

Operation 1 - set all formal input parameters equal to their corresponding actual input parameters. Set all formal output parameters to their default value.

Jump to subroutine.

Operation 2 - set all actual output parameters equal to their corresponding formal output parameters. Set all formal input parameters to their default value.

The passing of parameters using the method of call by reference in Basic programs is neither as simple or as common as the call by value and result method. The principle of call by reference is that the subroutine and main program are given the address of the parameter rather than its value. On the PET this means that values are stored and read from a protected area of memory using the PEEK and POKE commands. Call by reference has two virtues. First it allows the easy

incorporation of machine code subroutines within a Basic program. Secondly it allows parameters to be passed between programs in a suite, where normally all variables would be erased by the process of loading a program. This method does however pose several problems. Firstly all variables must have a fixed length and format, since a set number of locations starting at a specific address will be allocated to each variable when the program is written. Secondly the conversion of variables from strings or numeric variables to POKE values and vice versa requires in many cases fairly lengthy Basic code. This problem does not effect numeric variables with values of less than 256 since these can be stored as a single byte. The passing of parameters by reference is thus ideal for use with status flags or counters.

Debugging.

Once a program has been written it must be thoroughly tested and debugged before it is incorporated into the suite. Every permutation of input or output should be checked against the expected input or output on the flow chart and specifications. Files should be checked that data is stored in the correct format, and that data read by the program from file is correct. One way of checking programs is to insert break points (a STOP command) into the program at points where one wishes to check the contents of variables etc. A useful feature of PET interpreter Basic is the ability to check the contents of variables after a break by printing the variable in the immediate mode. This way one can work through a program checking on the logic flow. To check file handling, test files can be created using the input program in the suite (this is the first program in the suite to be written). If the program being written is the input program, then the only existing files required will be parameter files, these can be created by small test programs written for the purpose. Similar test programs can be used to check the contents of files written by the program.

The process of debugging a program can be the most tedious and time consuming part of writing a program. It must be noted that the amount of time spent debugging is very often inversely proportional to the time spent on the initial planning and specification of the program. The process of debugging is also made easier if REM statements are liberally used throughout the program. One can then easily determine the exact function of a block of code without having to trace it through the whole code.

The Program Suite.

Once all the programs in the suite have been written and debugged, they can be integrated to form the program

suite by means of a menu program. A menu program is simply a means of selecting and loading a program without having to use the normal load command. Menus are also used within programs to select one of a range of functions or options. Having linked the programs together via the menu, the complete suite should be tested, to ensure that there are no problems in transferring data or parameters between programs. As with debugging individual programs, the chances of faults appearing at this stage are considerably reduced if sufficient care was taken in the program design and specification stages.

Notes:

The majority of subroutines have been written to conform to certain standards. First, all subroutines have line numbers above 20000. Second, major parameters passed between the subroutine and the main program usually begin with the character P. Third, wherever possible subroutine calls within subroutines have been avoided (the principle exception to this is the subroutine, CURSORCONT).

To make the reader's use and comprehension of the subroutine's function, and integration into a program easier, the majority of subroutines have been given sample calling routines. These routines usually start at line 1000 and end prior to line 2000. Line numbers before 1000 usually contain variables and constants used by the subroutine.

All the subroutines and programs in this book are available on disk, copies may be obtained from Computabits Ltd, P.O. Box 13, Yeovil, Somerset. price £10.00.

DATA INPUT

Data entry poses several problems when writing a program. Firstly the normal INPUT command's inability to deal with commas, colons or null entries. Secondly the need to be able to prevent the wrong keys being pressed or the wrong variable type entered (eg. an alpha character entered in a numeric input). Thirdly numeric and date inputs should be checked for incorrect entry; the facility should also exist for controlling the maximum and minimum number of characters in any entry. Lastly data input should be easy for the operator to use, incorporating full editing both of the current entry and preceding entries. These problems can be overcome by using special data input subroutines.

In all these data entry subroutines written in Basic, the INPUT command is replaced by the GET command. This allows each character of the input to be tested, to see if it is of the correct variable type or is one of the command keys, before it is added to the data already input. All variables are therefore input as string variables. The only problem with using the GET command is that in machines using Basic 1.00, 2.00 or 3.00 the occasional delay caused by garbage collection can be a nuisance. This is more frequent when using GET for input owing to the repeated concatenation of strings. Despite this the use of GET gives the programmer far greater control over data input.

In order to give the operator editing capability certain keys must be given command functions. The following is a proposed standard for key functions.

Stop key.

The stop key is disabled at the beginning of a program with the command: POKE 144,49. At the end of the program the stop key should be reenabled with POKE 144,46.

Delete key.

This key in the unshifted mode will have the function of deleting the entire entry with the prompt being returned to the first character in the field.

Right Arrow.

This key in the unshifted mode deletes the last character entered and places the prompt back one character.

Cursor Up.

The shifted cursor-up key has the function of aborting the existing entry field and returning to the preceding entry on the screen (if any). The preceding field is erased and the

prompt placed in the first character position. The aborted field contains a null string entry.

Cursor Down.

The unshifted cursor-down key aborts the existing entry field and places the prompt in the first character of the next field on the screen (if any). The aborted field contains a null string entry.

Return.

Pressing return will accept an entry, provided it is within certain parameters and passes any error checking procedures. The parameters used are maximum and minimum entry length.

Question mark.

This is an optional command key which can be used to display operator help instructions. These instructions are displayed on the bottom two lines of the display.

The bottom two lines (24 & 25) are reserved for error messages, prompts, or help messages related to the line currently being input. On all inputs an underline showing the maximum input length is used as an input prompt.

ALPHAINPUT

This subroutine inputs a data string with a maximum length set by parameter P3 and a minimum length by P4. The subroutine allows the entry to be edited using the left arrow to delete the last entered character and the delete key to delete the entire entry. The entry is accepted and the subroutine exits either when the return key is pressed and the data exceeds the minimum entry length or when the data input equals the maximum data length.

Parameters used:

P3 - maximum length of input data string P\$, an input prompt line P3 characters long is drawn by lines 28010-28012.

P4 - minimum length of input data string P\$, one can not exit from the subroutine until an entry of P4+ characters has been input.

P\$ - character string input by subroutine.

DIGITINPUT

This is another version of the preceding subroutine, the difference being that the only characters accepted for data input are numeric variables, decimal point, and minus sign. The subroutine has the same input and output parameters and

editing functions as Alphainput.

ADDRESSIN

Whereas ALPHAINPUT is a general purpose input subroutine for alphanumeric variables ADDRESSIN is specifically designed to input a name and address. The subroutine will input a name and address of five lines each line having up to 30 characters. Each line is automatically terminated and the next line commenced by pressing the comma key, except for line 5 where entering a comma has no effect. This allows the name and address to be entered without having to press return after each line since the lines are naturally terminated with a comma. Full editing on a line is allowed using either delete or left arrow, also the current line can be aborted and a previous line reentered by using the cursor-up key. All alphanumeric characters are accepted as valid except command characters and spaces where they are the first character on a line. The address entry is terminated and the subroutine exits only when the key sequence - full stop - return - is pressed.

Parameters used:

P\$(5) - input string array, one element to each line on the address.

YES/NO

This is another special purpose input subroutine with the very simple function of inputting a yes or no answer to a question. Pressing Y for yes will set output parameter P to 1 and YES will be printed on the screen. Pressing N will set P to 2 and NO will be displayed.

Parameters used:

P - yes/no flag: 1=yes and 2=no.

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO INPUT AN ALPHA STRING
1003 REM *MINIMUM LENGTH 3 CHARACTERS
1004 REM *MAXIMUM 5 CHARACTERS.
1009 REM *****
1010 P3=5:P4=3:REM **LENGTH MAX/MIN**
1020 GOSUB28000
1030 PRINT:PRINT:PRINTP$:REM **PRINT INPUT STRING **
1100 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
28000 REM *****
28002 REM *SUBROUTINE TO INPUT A
28004 REM *DATA STRING
28006 REM *****
28008 P$=""
28009 REM ***PRINT PROMPT LINE***
28010 FORQ=1TOP3:PRINT"_":NEXTQ:PRINT"! ";
28012 FORQ=1TOP3+1:PRINT"||":NEXTQ
28014 REM ***GET CHARACTER***
28016 GETA$:IFA$=""GOTO28016
28020 REM ***TEST FOR COMMAND KEYS***
28022 IFA$=CHR$(13)GOTO28068
28024 IFA$="←"THEN28040
28026 IFA$<>CHR$(20)GOTO28052
28030 REM ***DELETE ENTRY***
28032 PL=LEN(P$):FORQ=1TOPL:PRINT"||":NEXT
28034 GOTO28008
28038 REM ***DELETE LAST CHARACTER***
28040 IFP$=""GOTO28014
28042 IFLEN(P$)=1THENP$="":GOTO28046
28044 P$=LEFT$(P$,LEN(P$)-1)
28046 A$="||":GOTO28064
28050 REM ***VALID ALPHA CHARACTER?***
28052 IFA$<" "GOTO28014
28054 IFA$>"Z"AND A$<"||"GOTO28014
28058 REM ***ENTRY TOO LONG?***
28060 IFLEN(P$)=P3GOTO28070
28062 P$=P$+A$
28064 PRINTA$;
28066 GOTO28014
28067 REM ***ENTRY TOO SHORT?***
28068 IFLEN(P$)<P4GOTO28014
28070 RETURN
READY.

```

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO INPUT A NUMERIC STRING
1003 REM *MINIMUM LENGTH 3 DIGITS
1004 REM *MAXIMUM LENGTH 5 DIGITS
1009 REM *****
1010 P3=5:P4=3:REM **MAX/MIN LENGTH**
1020 GOSUB28100
1030 REM **CONVERT NUMERIC STRING TO NUMERIC VARIABLE**
1040 P=VAL(P$)
1050 PRINT:PRINT:PRINTP:REM **PRINT NUMERIC INPUT**
1060 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
28100 REM *****
28102 REM *SUBROUTINE TO INPUT A
28104 REM *NUMERIC STRING
28106 REM *****
28108 P$=""
28110 REM ***PRINT PROMPT LINE***
28112 FORQ=1TOP3:PRINT"_":NEXTQ:PRINT" ";
28114 FORQ=1TOP3+1:PRINT"|||":NEXTQ
28116 REM ***GET CHARACTER***
28118 GETA$:IFA$=""GOTO28118
28120 REM ***TEST FOR COMMAND KEYS***
28122 IFA$=CHR$(13)GOTO28166
28124 IFA$="←"GOTO28136
28126 IFA$<>CHR$(20)GOTO28146
28128 REM ***DELETE ENTRY***
28130 PL=LEN(P$):FORQ=1TOPL:PRINT"|||":NEXT
28132 GOTO28108
28134 REM ***DELETE LAST CHARACTER***
28136 IFP$=""GOTO28118
28138 IFLEN(P$)=1THENP$="":GOTO28142
28140 P$=LEFT$(P$,LEN(P$)-1)
28142 A$="|||":GOTO28160
28144 REM ***VALID NUMERIC CHARACTER?***
28146 IFA$="."THEN28158
28148 IFA$="-"ANDP$=""THEN28158
28150 IFA$<"0"GOTO28118
28152 IFA$>"9"GOTO28118
28154 REM ***ENTRY TOO LONG?***
28156 IFLEN(P$)=P3GOTO28166
28158 P$=P$+A$
28160 PRINTA$:
28162 GOTO28118
28164 REM ***ENTRY TOO SHORT?***
28166 IFLEN(P$)<P4THENGOTO28118
28168 RETURN
READY.

```

```

100 LNE$="XXXXXXXXXXXXXXXXXXXXXXXXXXXX"
110 DIM P$(5)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO INPUT AN ADDRESS.
1009 REM *****
1010 PRINT"J";
1020 GOSUB 28300
1030 REM **PRINT ADDRESS INPUT FROM ARRAY P$(Q)**
1040 PRINT"J";
1050 FOR Q=1TO5
1060 PRINTP$(Q)
1070 NEXTQ
1080 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
28300 REM *****
28302 REM *ADDRESS INPUT SUBROUTINE
28304 REM *****
28306 PRINT"XXXXXXXXXXXXXXXXXXXXADDRESS INPUT"
28308 REM ***LOOP FOR 5 ADDRESS LINES***
28310 FORQP=1TO5
28312 REM ***INITIALISE***
28314 P$(QP)=""
28316 REM ***HOME CURSOR & MOVE DOWN TO LINE START***
28318 REM ***THEN INDENT 5 SPACES & PRINT PROMPT LINE***
28320 PRINT"J";
28322 PRINTLEFT$(LNE$,7+QP*2);
28324 PRINT" ";
28326 FORQL=1TO30:PRINT"_";:NEXTQL
28328 FORQL=1TO30:PRINT" ";:NEXTQL
28330 REM ***GET CHARACTER***
28332 GETA$:IFA$=""GOTO28332
28334 REM ***TEST FOR COMMAND KEYS***
28336 REM ***IF RETURN + . THEN END ENTRY***
28338 IFA$=CHR$(13)ANDRIGHT$(P$(QP),1)=".":THENQP=5:GOTO28404
28340 REM ***IF , THEN NEXT LINE***
28342 IFA$=","THEN28396
28344 REM ***IF CURSOR UP THEN GOTO PREVIOUS LINE***
28346 IFA$=CHR$(145)THEN28372
28348 REM ***IF ← THEN DELETE LAST CHARACTER***
28350 IFA$="←"THEN28362
28352 REM ***IF DEL THEN DELETE CURRENT LINE***
28354 IFA$<>CHR$(20)GOTO28378
28356 REM ***DELETE CURRENT LINE***
28358 GOTO28314
28360 REM ***DELETE LAST CHARACTER***
28362 IFP$(QP)=""GOTO28332
28364 IFLEN(P$(QP))=1THENP$(QP)="":GOTO28368
28366 P$(QP)=LEFT$(P$(QP),LEN(P$(QP))-1)
28368 A$="|||":GOTO28392
28370 REM ***GO BACK TO PREVIOUS LINE***
28372 IFQP>1THENQP=QP-1
28374 GOTO28314

```

```

28376 REM   ***VALID CHARACTER INPUT?***
28378 IFA$<" "GOTO28332
28380 IFA$>"Z"AND A$<" "GOTO28332
28382 REM   ***DELETE LEADING SPACES***
28384 REM   ***CHECK LINE NOT TOO LONG***
28386 P$(QP)=P$(QP)+A$
28388 IF LEN(P$(QP))>30 THEN 28332
28390 IF LEFT$(P$(QP),1)=" " THEN P$(QP)=RIGHT$(P$(QP),LEN(P$(QP))-1):GOTO28332
28392 PRINT A$;:GOTO28332
28394 REM   ***IF NOT LAST LINE THEN ADD LAST CHAR & GOTO NEXT***
28396 IF QP<5 THEN PRINT A$:P$(QP)=P$(QP)+A$:GOTO28404
28398 REM   ***IF LAST LINE JUST ADD***
28400 GOTO28386
28402 REM   ***NEXT LINE***
28404 NEXT QP
28406 PRINT:PRINT:PRINT:PRINT
28408 REM   ***ADDRESS CORRECT? YES THEN RETURN***
28410 REM   ***NO THEN REENTER ADDRESS***
28412 PRINT"      ENTRY CORRECT YES OR NO ?"
28414 GET A$:IFA$=""GOTO28414
28416 IFA$="Y"THEN28422
28418 IFA$="N"THEN PRINT"J":GOTO28300
28420 GOTO28414
28422 RETURN
READY.

```

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *FOR YES/NO REPLY.
1009 REM *****
1010 GOSUB 28200
1020 REM **CONDITIONAL BRANCH ON VALUE OF P**
1030 ON P GOTO 1500,1700
1500 PRINT"THE REPLY WAS YES"
1510 END
1700 PRINT"THE REPLY WAS NO"
1710 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
28200 REM *****
28202 REM *SUBROUTINE TO INPUT YES
28204 REM *NO REPLY
28206 REM *****
28208 PRINT"YES OR NO ?";
28210 GET A$:IFA$=""GOTO28210
28212 IFA$="Y"THENPRINT"■■■■■■■■■■YES"      ":P=1:GOTO28218
28214 IFA$="N"THENPRINT"■■■■■■■■■■NO"       ":P=2:GOTO28218
28216 GOTO28210
28218 RETURN
READY.

```

REDUCING INPUT ERRORS

Data input errors are the primary cause of most so-called computer errors, such as the million pound electricity bill. Procedures can be adopted which greatly reduce the chance of incorrect numerical entry. The commonest cause of numerical data entry error is the misinterpretation of handwriting. This is overcome by adopting a standard method of writing. The letter "I" must be 'topped and tailed' to avoid its being mistaken for a numeric "1", zero is crossed to differentiate it from the letter "O", "Z" is crossed to avoid being mistaken for a "2" and "7" may also be crossed so that it can not be mis-read for a "1".

Mis-reading of handwriting is only one source of errors in numeric data entry. The types of error can be classified as:

Insertion - an extra character is added to a data item, the number 26195 may gain a 1 to become 261195.

Omission - a character is left out of a data item, the number 312263 may lose a 2 to become 31263.

Transcription - this is mis-reading the data, the 7 in 27620 may be misread as a 1 causing the number to become 21620.

Transposition - characters change position, 28140 may become 21840.

The final result of any of these errors is not only a corruption of data but in practice could mean a customer receiving the wrong goods or being billed incorrectly.

To overcome these errors one must try to ensure when writing a program that data will be read, written and stored accurately. One way of detecting errors in numerical data items is the use of a check digit. A check digit is produced by performing a calculation on the number and appending the result to the data item. Whenever a number is input to the computer the check digit is recalculated and checked against the input check digit. If there is an error then the two check digits will not match and the input can be rejected. The incorporation of check digits is particularly useful with part or account numbers.

There are many ways of calculating check digits, the methods used to ensure a 100% error detection rate usually involve a weighting technique. Such a method is used in the following subroutine, where each digit in the number is weighted, starting with the least significant digit the weight being incremented by one. Using the part number 72946 as an example and calculating the check digit for it thus:

$$\begin{array}{r}
 7 \ 2 \ 9 \ 4 \ 6 \\
 \begin{array}{l}
 \text{---} \times 2 = 12 \\
 \text{---} \times 3 = 12 \\
 \text{---} \times 4 = 36 \\
 \text{---} \times 5 = 10 \\
 \text{---} \times 6 = 42
 \end{array} \\
 \hline
 + 112
 \end{array}$$

This total is then divided by 11 and the remainder noted:

$$112/11 = 10 \text{ remainder } 2$$

The remainder is subtracted from 11 giving the check digit 9. Using this algorithm, check digits will be produced with a range between 0 and 11. Since most numbers using check digits will be of fixed length we can ensure that the number always has two digits simply by adding 10 to the check digit.

Parameters used:

P\$ - numeric input for which the check digit is to be calculated.

PC\$ - check digit calculated from the value in P\$.

Notes: line 25014 is used to detect negative numbers, these then have check digits in the range 50-61, this line can be omitted if no negative numbers are used. When using fractional numbers note that the check digit calculation will not differentiate between fractions smaller than .5. To overcome this the input variable should be multiplied by say 100 to give a two decimal place accuracy.

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO CREATE CHECK DIGIT FOR
1003 REM *NUMBER.
1009 REM *****
1010 INPUT"INPUT NUMBER";P$
1020 GOSUB25000
1030 PRINT:PRINT:PRINT"CHECK DIGIT IS";PC$
1040 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25000 REM *****
25002 REM *SUBROUTINE TO CREATE A CHECK
25004 REM *DIGIT FOR VALUE IN P$
25006 REM *****
25008 REM **FOR INCREASED RESOLUTION MULTIPLY P$ BY 100**
25010 A=100*VAL(P$):BD=10
25012 REM **IF NEGATIVE NUMBER THEN ADD 50 TO CHECK DIGIT**
25014 IFLEFT$(P$,1)="-"THEN A=VAL(RIGHT$(P$,LEN(P$)-1)):BD=50
25016 A1=0
25018 A2=2
25020 B=INT(A/10)
25022 A1=A1+(A-(B*10))*A2
25024 IFB=0THEN25032
25026 A=B
25028 A2=A2+1
25030 GOTO25020
25032 AC=A1-(INT(A1/11)*11)
25034 AC=11-AC
25036 AC=INT(AC)+BD
25038 REM **CHECK DIGIT FOR VALUE IN P$ IS PC$**
25040 PC$=STR$(AC)
25042 RETURN
READY.

```

DATE INPUT VERIFICATION AND STORAGE

Dates comprise a special class of input which is always prone to operator error unless validation techniques are used. An example is entering the date as the 31st when there are only 30 days in that month. The date validation program must be able to determine how many days there are in a given month and whether it is a leap year or not. Another problem is that there are many different ways of inputting a date, all being equally valid.

For calculation purposes, and for compactness when stored in a data file, dates can be converted into a special numeric form representing the number of days since 1/1/72. This allows comparison of dates and calculation of date differences.

DATEINPUT

This subroutine will validate a date input as variable P\$ with the format DDMMYY (ie: 2nd Feb 80 input as 020280). Lines 25120 to 25124 dissect the date string to give simple numeric variables for day, month and year. Line 25130 checks that these values are within certain general limits, if not then error flag variable P6 is set. Line 25132 calculates if the current year is a leap year, and line 25134 checks that there are not too many days in the month using AL\$ in a leap year and AM\$ in an ordinary year.

Parameters used:

P\$ - date input string in format DDMMYY with a length of 6 characters.

P6 - error flag if date in P\$ is invalid.

PD - numerical days in date.

PM - numerical month in date.

PY - numerical year in date.

DATEIN2

This subroutine is a more sophisticated and general purpose date validation subroutine. The date is input as string P\$ but can have a variable format - DDMMYY, DD-MM-YY, DD/MM/YY, DD-MMM-YY, or D-MMM, (e.g. 020280, 02-02-80, 02/02/80, 02-FEB-80, or 2-FEB). If no year is input with the format D-MMM then the year stored in the program as PY\$ in line 25508 is automatically inserted. Lines 25514 to 25534 determine which format has been used and convert that format to a standard internal format used by the subroutine for its validation procedures. If the input contains an alpha month

then lines 25540 to 25546 check the three character alpha month against valid alpha months stored as AM\$ to produce a numerical month value. If the input alpha month is invalid then error flag P is set to 1 and the subroutine exited. Lines 25552 to 25556 dissect the internal date string to produce numerical values for day month and year. The date validation is done by lines 25562 to 25568. Any errors will set the error flag P to 1 and exit from the subroutine.

Parameters used:

P\$ - date input string, length can be between 5 and 9 characters, format is variable.
P - error flag set to 1 if there is an error in the input variable P\$.
PD - numerical days in date.
PM - numerical month in date.
PY - numerical year in date.

DAYDATE1

This subroutine has not only the function of validating a date input it also converts the input format into an output format. The format used for input variable P\$ is either DDMMM or DDMMYY (e.g. 2FEB or 2FEB80). P\$ for this example is output by the subroutine as SATURDAY 2ND FEBRUARY 1980. Any error in the input causes the error flag P6 to be set to 1 and the subroutine exited. Line 25418 checks that the number of days in the date is within allowed limits. The presence of a year in the input is detected by lines 25418-25420. If there is no year in the input date then the default year stored as PY in line 25410 is used. Lines 25426 to 25430 convert the alpha month in the input into a numerical value. If there is an error in the input then P6 is set to 1 and the routine exits. The day of the week is calculated by lines 25436-25440. Lines 25446-25456 construct the output string P\$.

Parameters used:

P\$ - input string from 4 to 7 characters long.
P\$ - output string.
PD - numerical days in date.
PM - numerical month in date.
PY - numerical year in date.
P6 - error flag, error =1.

DATECONVERT

The function of this subroutine is to convert a date input as P\$ with the format DDMMYY into a numeric value representing the number of days between the input date and 1/1/72. Lines 25214-25218 dissect P\$ to give numeric days, months, and years in the date. This section of the subroutine could be omitted if these values already exist. The output is given as both numeric and string variables PN and PN\$.

Parameters used:

P\$ - input string with date in format DDMMYY.
PN - numeric days since 1/1/72.
PN\$- string variable equivalent of PN.

DATERESTORE

The numeric internal date format of days since 1/1/72 is converted back into a normal format DD/MM/YY by this subroutine. The input parameter is PN\$. Change line 25316 to AD=PN if input parameter is a numeric variable. The output parameter is P\$.

Parameters used:

PN\$ - input string of days between date and 1/1/72.
P\$ - output string with date in format DD/MM/YY.

```

1000 REM*****
1001 REM*EXAMPLE OF SUBROUTINE CALL
1002 REM*TO INPUT AND VALIDATE DATE.
1009 REM*****
1010 INPUT"INPUT A DATE -DDMMYY -";P$
1020 GOSUB25100
1030 REM ***IF INVALID DATE REENTER***
1040 IFP6=1THEN GOTO1010
1050 PRINT:PRINT:PRINT"DAY-";PD;"  MONTH-";PM;"  YEAR-19";PY
1060 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25100 REM *****
25102 REM *DATE INPUT AND VALIDATION
25104 REM *DATE INPUT IN FORMAT -DDMMYY
25106 REM *AS VARIABLE P$.
25108 REM *****
25110 AM$="312831303130313130313031"
25112 AL$="312931303130313130313031"
25114 P6=0
25116 REM ***DISSECT INPUT STRING***
25118 REM ***TO GIVE NUMERIC DATES***
25120 PD=VAL(LEFT$(P$,2))
25122 PM=VAL(MID$(P$,3,2))
25124 PY=VAL(RIGHT$(P$,2))
25126 REM ***CHECK THAT THE DATE INPUT IS WITHIN PARAMETERS***
25128 REM
25130 IFPM<10RPM>12ORPY<70ORPY>85ORPD<01ORPD>31THENP6=1
25132 AR=1900+PY: IFINT(AR/4)=AR/4THENAM$=AL$
25134 IFPD>VAL(MID$(AM$,PM*2-1,2))THENP6=1
25136 RETURN
READY.

```

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO INPUT AND VALIDATE A DATE.
1009 REM *****
1010 INPUT"INPUT A DATE -";P$
1020 GOSUB25500
1030 REM **IF INVALID THEN REENTER**
1040 IFP=1THENGOTO1010
1050 PRINT:PRINT:PRINT"DAY-";PD;" MONTH-";PM;" YEAR- 19";PY
1060 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25500 REM*****
25502 REM*DATE INPUT AND VALIDATION
25504 REM*****
25506 AM$="JANFEBMARAPR MAYJUNJUL AUGSEP OCTNOVDEC"
25508 PY$="-80":P=0
25510 REM ***DETERMINE FORMAT OF ***
25512 REM ***DATE INPUT***
25514 AL=LEN(P$)
25516 IFAL<5THENP=1:RETURN
25518 IFMID$(P$,4,1)>="A"THEN25530
25520 IFAL=5ANDVAL(MID$(P$,3,1))=0THENP$=P$+PY$:GOTO25552
25522 IFAL=5ORAL=7THENP$="0"+P$
25524 IFAL>6THEN25552
25526 P$=LEFT$(P$,2)+"-"+MID$(P$,3,2)+"-"+RIGHT$(P$,2)
25528 GOTO25552
25530 IFAL<7THENP$=P$+PY$
25532 IFAL=8THENP$="0"+P$
25534 IFAL=5THENP$="0"+P$
25536 REM ***DETERMINE MONTH NUMBER***
25538 REM ***FOR ALPHA MONTH INPUT***
25540 A0$=MID$(P$,4,3)
25542 FORPM=1TO12
25544 IFA0$=MID$(AM$,3*PM-2,3)THEN25554
25546 NEXTPM:P=1:RETURN
25548 REM ***DISSECT INPUT STRING***
25550 REM ***TO GIVE NUMERIC DATES***
25552 PM=VAL(MID$(P$,4,2))
25554 PD=VAL(LEFT$(P$,2))
25556 PY=VAL(RIGHT$(P$,2))
25558 REM ***CHECK THAT DATE INPUT***
25560 REM ***IS WITHIN PARAMETERS***
25562 IFPD=0ORPM=0ORPY=0THENP=1:RETURN
25564 IFPD>31ORPM>12THENP=1:RETURN
25566 IFPD=29>VAL(MID$("202121221212",PM,1))THENP=1:RETURN
25568 IFPM=2ANDPD=29AND(PYAND3)<>0THENP=1
25570 RETURN
READY.

```

```

200 DIMAM$(12),AI$(12),AW$(7)
202 AW$(0)="TUESDAY":AW$(1)="WEDNESDAY"
204 AW$(2)="THURSDAY":AW$(3)="FRIDAY"
206 AW$(4)="SATURDAY":AW$(5)="SUNDAY"
208 AW$(6)="MONDAY"
210 AM$(1)="JANUARY":AM$(2)="FEBRUARY"
212 AM$(3)="MARCH":AM$(4)="APRIL"
214 AM$(5)="MAY":AM$(6)="JUNE":AM$(7)="JULY"
216 AM$(8)="AUGUST":AM$(9)="SEPTEMBER"
218 AM$(10)="OCTOBER":AM$(11)="NOVEMBER"
220 AM$(12)="DECEMBER"
222 AI$(1)="JAN":AI$(2)="FEB":AI$(3)="MAR"
224 AI$(4)="APR":AI$(5)="MAY"
226 AI$(6)="JUN":AI$(7)="JUL":AI$(8)="AUG"
228 AI$(9)="SEPT":AI$(10)="OCT"
230 AI$(11)="NOV":AI$(12)="DEC"
300 REM
400 REM
500 REM
600 REM
700 REM
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO INPUT DATE AND PRINT DATE
1003 REM *IN FULL FORM.
1009 REM *****
1010 INPUT"INPUT A DATE - ";P$
1020 GOSUB25400
1030 REM **IF INVALID THEN REENTER**
1040 IFP6=1THEN GOTO1010
1050 PRINT:PRINT:PRINTP$
1060 PRINT:PRINT:PRINT"DAY-";PD;" MONTH-";PM;" YEAR-";PY
1070 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25400 REM*****
25402 REM*SUBROUTINE TO INPUT AND VALIDATE
25404 REM*THE DATE THEN CALCULATE THE DAY
25406 REM*OF THE WEEK
25408 REM*****
25410 AM=0:AZ=0:AX=0:P6=0:PY=1980
25412 REM ***DISSECT AND VALIDATE***
25414 REM ***DATE INPUT STRING***
25416 AD=VAL(P$):AL=LEN(P$)
25418 IFAD<10RAD>31THENP6=1:GOTO25458
25420 AX$=RIGHT$(P$,2):AY=VAL(AX$)
25422 IFAY>0THENAY=AY-PY+1900
25424 AX=3+(AD<10):AM$=MID$(P$,AX,3)
25426 FORQ=1TO12
25428 IFAM$=AI$(Q)THENAM=Q:GOTO25436
25430 NEXTQ:P6=1:GOTO25458
25432 REM ***CALCULATE DAY OF WEEK***
25434 REM ***FROM NUMERIC DATE VARIABLES***
25436 Q=12:AS=AD:AX=AY+PY:AS=AS+AX*365

```

```

25438 IFAM=>3THENAS=AS-INT(AM*.4+2.3):AX=AX+1
25440 AS=AS+INT(AM*31+(AX-1)/4):AW=AS-INT(AS/7)*7
25442 REM ***CONVERT DATE VARIABLES***
25444 REM ***INTO OUTPUT STRING***
25446 AT$=".TH."
25448 IFAD=1ORAD=21ORAD=31THENAT$=".ST."
25450 IFAD=2ORAD=22THENAT$=".ND."
25452 IFAD=3ORAD=23THENAT$=".RD."
25454 P$=" "+AW$(AW)+STR$(AD)+AT$+" "+AM$(AM)+STR$(AY+PY)
25456 PD=AD:PM=AM:PY=AY+PY
25458 RETURN
READY.

```

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL TO
1002 REM *CONVERT A DATE INTO INTERNAL
1003 REM *FORMAT
1009 REM *****
1010 INPUT "INPUT DATE - DDMMYY - "; P$
1020 GOSUB 25200
1030 PRINT:PRINT:PRINT "INTERNAL FORMAT DATE IS "; PN$
1040 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25200 REM *****
25202 REM *DATE CONVERSION TO INTERNAL
25204 REM *FORMAT OF DAYS SINCE 1/1/72
25206 REM *INPUT IN P$ AS (DDMMYY),
25208 REM *OUTPUT IN PN$.
25210 REM *****
25212 A1$="000031059090120151181212243273304334"
25214 A1=VAL(LEFT$(P$,2))
25216 A2=VAL(MID$(P$,3,2))
25218 A3=VAL(RIGHT$(P$,2))
25220 AD=A1+(A3-72)*365+INT((A3-69)/4)+VAL(MID$(A1$,3*A2-2,3))
25222 IF A2>2 AND (1900+A3)/4=INT((1900+A3)/4) THEN AD=AD+1
25224 PN=AD:PN$=STR$(PN)
25226 RETURN
READY.

```

```

25200 REM *****
25202 REM *DATE CONVERSION TO INTERNAL
25204 REM *FORMAT OF DAYS SINCE 1/1/72
25206 REM *INPUT AS VARIABLES PD,PM,PY
25208 REM *(DAY,MONTH,YR) OUTPUT IN PN$.
25210 REM *****
25212 A1$="000031059090120151181212243273304334"
25220 A1=PI+(PY-72)*365+INT((PY-69)/4)+VAL(MID$(A1$,3*PM-2,3))
25222 IF PM>2 AND (1900+PY)/4=INT((1900+PY)/4) THEN AD=AD+1
25224 PN=AD:PN$=STR$(PN)
25226 RETURN
READY.

```

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL TO
1002 REM *CONVERT INTERNAL FORMAT DATE
1003 REM *INTO STANDARD DATE FORMAT-DDMMYY.
1009 REM *****
1010 INPUT"INTERNAL FORMAT DATE EG:3207 ";PN$
1020 GOSUB25300
1030 PRINT:PRINT:PRINT"CORRESPONDING DATE IS - ";P$
1040 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
25300 REM *****
25302 REM *DATE RECONSTRUCTION FROM
25304 REM *INTERNAL FORMAT INPUT AS PN$
25306 REM *DATE OUTPUT AS P$ IN FORM
25308 REM *(DD/MM/YY).
25310 REM *****
25312 A1$="000031059090120151181212243273304334"
25314 A2=34
25316 AD=VAL(PN$)
25318 A3=INT(AD/365)
25320 A1=AD-(A3*365)-INT((A3+3)/4)
25322 IF A1<0 THEN 25328
25324 A3=A3-1:A1=A1+365
25326 IF INT(A3/4)=A3/4 THEN A1=A1+1
25328 A5=0:IF INT(A3/4)=A3/4 THEN A5=1
25330 A4=VAL(MID$(A1$,A2,3))
25332 IF A4>31 THEN A4=A4+A5
25334 IF A1<A4 THEN A2=A2-3:GOTO 25330
25336 A1$=STR$(A1-A4)+"/"
25338 A2$=STR$((A2+2)/3)+"/"
25340 P$=RIGHT$("0"+RIGHT$(A1$,LEN(A1$)-1),3)
25342 P$=P$+RIGHT$("0"+RIGHT$(A2$,LEN(A2$)-1),3)+RIGHT$(STR$(A3+72),2)
25344 RETURN
READY.

```

SCREEN FORMAT AND DISPLAY

The subroutines in this section are all designed to control the way data is output to the screen.

CURSORCONT

This cursor control subroutine can be the basis of any screen data output procedure. The subroutine places the cursor at a location on the screen defined by the co-ordinates of line and column numbers. The position of the cursor is then the starting position for subsequent printing. Lines 29810, 29818 and 29820 are used to remove any print on the screen for 40 characters in front of the cursor. This feature is useful if the cursor is at the beginning of a line since it will delete all the existing contents of that line. If not required then these lines of the subroutine can be omitted.

Parameters used:

COL - this value should be set to the column number (between 1 and 40) at which the cursor is to be placed.
LNE - this value should be set to the line number (between 1 and 25) on which the cursor is to be placed.

WARNING

When an error situation arises during a program, because of operator error, data file read or write errors, etc; this subroutine can be used to signal the fact to the operator. The subroutine calls the subroutine Cursorcont in line 30106 to place the cursor at the beginning of line 25 (the bottom line of the screen). A flashing message "WARNING" is displayed at the beginning of the line (Note: AW=LOG(m)generates a delay between flashes). A message stored as MES\$ is then displayed on the rest of the line. The subroutine then halts until a key is pressed, upon which line 25 on the screen is erased and the subroutine exits.

Parameters used:

MES\$ - message to be displayed on line 25 indicating fault or error type, maximum length 32 characters.
SP\$ - string consisting of 40 spaces.

BORDER

This little subroutine draws a border around the screen (the character used for the border is the graphics &, but could be any character). No parameters are passed by this subroutine.

BORDER2

This subroutine also draws a border around the screen but in this case using a thin line. The screen is given a page heading in reverse field characters in the centre of line 1.

Parameters used:

P\$ - string for page heading, maximum length 40 characters.
SP\$ - string of 40 spaces.

DIGIT

One of the problems in displaying numerical values on the screen in tabular form is that they are naturally left hand justified and, unless they are integer values, can have a variable number of decimal places. This gives rise to a very ragged display, to look neat the numbers should be truncated to a set number of decimal places, usually two, and right hand justified so that the decimal points are aligned in straight columns. This subroutine performs both these functions. Lines 27026 and 27028 round the fractional part of the number up or down to two decimal places. Then lines 27036 to 27054 truncate the fractional part of the string representation of the number to two decimal places, or add .00 to the number if it is an integer. All numbers are now in string format with the fractional component of two decimal places, the least significant digit of which has been rounded up or down, depending on the value of the number in the fractional third decimal place of the original number. Line 27056 adds spaces to the front of the numerical string so that all numbers output from the subroutine as variable P\$ have the same length. This is set by input parameter P.

Parameters used:

P\$ - input numerical string.
P - input parameter for length of output numerical string.

P\$ - output numerical string length P characters set to two decimal places.
SP\$- string of 40 spaces.

DISPLAYSRC and BDISP

DISPLAYSRC is a machine code program to perform various screen display functions, BDISP is a Basic loader version of DISPLAYSRC. This subroutine which is located in the top 2K of memory in a 32K machine is designed to perform several functions. First to draw horizontal or vertical lines on the screen of any length and starting at any position, using any valid character to construct the line. Secondly using the first two functions to draw a border around the screen and thirdly to reverse field a section of the screen or even the whole screen. Machine code is used to perform these functions because its higher speed of operation gives an almost instantaneous display. All these functions except the border require parameters to be poked into reserved memory locations. These parameters control starting position, character used, line length, and, with the reverse field k, the block dimensions. The starting position is the location in screen memory of the top right hand character of the line. This screen location is stored as two bytes, most significant(MSB) and least significant(LSB). These two location values must be calculated before a line is drawn or a block reversed. Screen lines should be numbered from 0 to 24 and columns from 1 to 40. The screen memory location can then be calculated by multiplying the line number of the starting location by 40 and adding to it the column number plus 32768, the result will be a value between 32768 and 33767. To obtain the MSB and LSB values for this number divide it by 256. The integer part of the result is the MSB value, the remainder is the LSB value. Thus if the starting position is line 4 column 10 the calculation is:

$$4*40 + 10 + 32768 = 32938$$
$$32938/256 = 128 \text{ remainder } 170$$

the MSB value is 128, LSB value 170

To draw a horizontal line the following Basic commands are required:

```
POKE 84, Start location LSB
POKE 85, Start location MSB
POKE 87, Line length
POKE 88, ASCII code for character used
SYS(31232)
```

To draw a vertical line:

- POKE 84, Start\location LSB
- POKE 85, Start position MSB
- POKE 87, Line length
- POKE 88, ASCII character code
- SYS(31246)

To draw a border

- SYS(31283)

To reverse field a block of the screen:

- POKE 84, Column number of start
- POKE 85, 128
- POKE 86, Line number of start
- POKE 87, Number of lines in block
- POKE 88, Number of columns in block
- SYS(31358)

22

5236

```

29900 REM *****
29902 REM *SUBROUTINE TO DRAW A BORDER
29904 REM *ARROUND THE SCREEN.
29906 REM *****
29908 PRINT "J": A$=" ": FOR I=1 TO 25: A$=A$+" ": NEXT
29910 B$=" ": FOR I=1 TO 40: B$=B$+" ": NEXT
29912 FOR I=1 TO 20: PRINT "J" TAB (20-I) LEFT$(B$, 2*I): NEXT
29914 PRINT "J": FOR I=1 TO 21: PRINT " " SPC (38) " ": NEXT: PRINT
29916 PRINT " " SPC (38) "J":
29918 FOR I=0 TO 19: PRINT "J" SPC (I) " " SPC (38-I*2) " ": NEXT
29920 RETURN
READY.

```

```

100 SP$=" "
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO DRAW BORDER WITH SCREEN
1003 REM *HEADING.
1009 REM *****
1010 PRINT "J";
1020 P$="EXAMPLE": REM **HEADING**
1030 GOSUB 29900
1040 PRINT "*****TEXT *****";
1050 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
29900 REM *****
29902 REM *SUBROUTINE TO DRAW BORDER
29904 REM *ARROUND THE SCREEN WITH A
29906 REM *REVERSE FIELD PAGE HEADING
29908 REM *TRANSFERED AS P$
29910 REM *****
29912 A=LEN(P$): A1=(40-A)/2
29914 A1$=LEFT$(SP$, A1)
29916 PRINT "J";
29918 PRINT A1$: PRINT " " P$ " "
29920 PRINT " "
29922 FOR X=32850 TO 33610 STEP 40: POKE X, 101: NEXT X
29924 FOR X=32886 TO 33686 STEP 40: POKE X, 101: NEXT X
29926 PRINT "*****"
29928 PRINT "J": RETURN
READY.

```

```

100 SP$="
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO DISPLAY FORMATTED NUMBERS
1009 REM *****
1010 INPUT"INPUT 5 NUMBERS - ";A$(1),A$(2),A$(3),A$(4),A$(5)
1020 PRINT:PRINT:PRINT
1030 P=12
1040 FORQ=1TO5
1050 P$=A$(Q)
1060 GOSUB27000
1070 PRINTP$
1080 NEXTQ
1090 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
27000 REM*****
27002 REM*SUBROUTINE TO ROUND NUMBERS
27004 REM*TO TWO DECIMAL PLACES AND
27006 REM*PAD WITH SPACES TO RIGHT
27008 REM*JUSTIFY COLUMNS. NUMBER INPUT
27010 REM*AND OUTPUT AS P$, OUTPUT
27012 REM*LENGTH P CHARACTERS.
27014 REM*****
27016 A$="":A1$="":A2$=""
27018 AN=VAL(P$)
27020 REM
27022 REM ***ROUND TO 2 PLACES***
27024 REM
27026 A1=- (INT(AN)-AN)
27028 A1=SGN(A1)*INT(ABS(A1*100)+.5)/100
27030 REM
27032 REM ***FORMAT NUMBER***
27034 REM
27036 A$=STR$(A1)
27038 IFA1<.01 THENA$="0.00"
27040 IFLEN(A$)=4 THEN27052
27042 IFLEN(A$)>4 THENA$=LEFT$(A$,4)
27044 IFLEN(A$)<4 THENA$=A$+"0"
27046 REM
27048 REM ***JUSTIFY***
27050 REM
27052 A$=RIGHT$(A$,3)
27054 A$=STR$(INT(AN))+A$
27056 P$=RIGHT$(SP$+A$,P)
27058 RETURN
READY.

```

LINE#	LOC	CODE	LINE
0001	0000		*****
0002	0000		SCREEN DISPLAY
0003	0000		SUBROUTINES
0004	0000		5/4/80
0005	0000		*****
0006	0000		
0007	0000		
0008	0000		
0009	0000		STARTL=\$54
0010	0000		STARTH=\$55
0011	0000		TEMP1=\$56
0012	0000		TEMP2=\$57
0013	0000		TEMP3=\$58
0014	0000		
0015	0000		
0016	0000		BEGIN=\$7A00
0017	0000		*=BEGIN
0018	7A00		
0019	7A00		HORIZONTAL LINE DRAW S/R
0020	7A00		
0021	7A00		DRAW HORIZONTAL LINE
0022	7A00		LENGTH TEMP2 USING
0023	7A00		CHARACTER WITH ASCII
0024	7A00		NUMERIC VALUE IN TEMP3
0025	7A00		
0026	7A00	98	HORIZ TYA
0027	7A01	48	PHA
0028	7A02	A4 57	LDY TEMP2
0029	7A04	A5 58	NEXTH LDA TEMP3
0030	7A06	91 54	STA (STARTL),Y
0031	7A08	88	DEY
0032	7A09	D0 F9	BNE NEXTH
0033	7A0B	68	PLA
0034	7A0C	A8	TAY
0035	7A0D	60	RTS
0036	7A0E		
0037	7A0E		VERTICAL LINE DRAWING S/R
0038	7A0E		
0039	7A0E		VERTICAL LINE LENGTH TEMP2
0040	7A0E		CHARACTER ASCII NUMERIC
0041	7A0E		VALUE IN TEMP3
0042	7A0E		
0043	7A0E	98	VERT TYA
0044	7A0F	48	PHA
0045	7A10	A4 57	LDY TEMP2
0046	7A12	A2 00	LDX #\$00
0047	7A14	A5 58	NEXTV LDA TEMP3
0048	7A16	81 54	STA (STARTL,X)
0049	7A18	20 21 7A	JSR NEXTL
0050	7A1B	88	DEY
0051	7A1C	10 F6	BPL NEXTV
0052	7A1E	68	PLA
0053	7A1F	A8	TAY
0054	7A20	60	RTS
0055	7A21		

LINE#	LOC	CODE	LINE
0056	7A21		;CALCULATE START OF NEXT SCREEN
0057	7A21		;LINE WITH ANY OFFSET
0058	7A21		;
0059	7A21	98	NEXTL TYA
0060	7A22	48	PHA
0061	7A23	18	CLC
0062	7A24	A9 28	LDA #\$28
0063	7A26	65 54	ADC STARTL
0064	7A28	85 54	STA STARTL
0065	7A2A	A9 00	LDA #\$00
0066	7A2C	65 55	ADC STARTH
0067	7A2E	85 55	STA STARTH
0068	7A30	68	PLA
0069	7A31	A8	TAY
0070	7A32	60	RTS
0071	7A33		;
0072	7A33		;S/R TO DRAW BORDER AROUND SCREEN
0073	7A33		;
0074	7A33	48	BORDER PHA
0075	7A34	98	TYA
0076	7A35	48	PHA
0077	7A36	8A	TXA
0078	7A37	48	PHA
0079	7A38	A9 A0	LDA #\$A0 ;CHARACTER USED FOR BORDE
0080	7A3A	85 58	STA TEMP3
0081	7A3C	A9 FF	LDA #\$FF ;START OF TOP
0082	7A3E	85 54	STA STARTL
0083	7A40	A9 7F	LDA #\$7F ;LINE
0084	7A42	85 55	STA STARTH
0085	7A44	A9 28	LDA #\$28 ;LINE LENGTH
0086	7A46	85 57	STA TEMP2
0087	7A48	20 00 7A	JSR HORIZ ;DRAW TOP LINE
0088	7A4B	A9 BF	LDA #\$BF ;START OF BOTTOM
0089	7A4D	85 54	STA STARTL
0090	7A4F	A9 83	LDA #\$83 ;LINE
0091	7A51	85 55	STA STARTH
0092	7A53	A9 28	LDA #\$28 ;LENGTH OF BOTTOM LINE
0093	7A55	85 57	STA TEMP2
0094	7A57	20 00 7A	JSR HORIZ ;DRAW BOTTOM LINE
0095	7A5A	A9 27	LDA #\$27 ;START OF RIGHT
0096	7A5C	85 54	STA STARTL
0097	7A5E	A9 80	LDA #\$80 ;LINE
0098	7A60	85 55	STA STARTH
0099	7A62	A9 17	LDA #\$17 ;LINE LENGTH
0100	7A64	85 57	STA TEMP2
0101	7A66	20 0E 7A	JSR VERT ;DRAW RIGHT LINE
0102	7A69	A9 00	LDA #\$00 ;START OF LEFT
0103	7A6B	85 54	STA STARTL
0104	7A6D	A9 80	LDA #\$80 ;LINE
0105	7A6F	85 55	STA STARTH
0106	7A71	A9 17	LDA #\$17 ;LINE LENGTH
0107	7A73	85 57	STA TEMP2
0108	7A75	20 0E 7A	JSR VERT ;DRAW LEFT LINE
0109	7A78	68	PLA
0110	7A79	AA	TAX

DISPLA5RC.....PAGE 0003

LINE#	LOC	CODE	LINE
0111	7A7A	68	PLA
0112	7A7B	A8	TAY
0113	7A7C	68	PLA
0114	7A7D	60	RTS
0115	7A7E		;
0116	7A7E		;S/R TO REVERSE FIELD BLOCK OF SCREEN
0117	7A7E		;
0118	7A7E		;START LINE # IN TEMP1
0119	7A7E		;# OF LINES IN BLOCK = TEMP2
0120	7A7E		;# OF COLUMNS IN BLOCK = TEMP3
0121	7A7E		;STARTL = COLUMN # OF START
0122	7A7E		;
0123	7A7E	48	BLOCK PHA
0124	7A7F	98	TYA
0125	7A80	48	PHA
0126	7A81	8A	TXA
0127	7A82	48	PHA
0128	7A83	A6 56	BLOCK1 LDX TEMP1
0129	7A85	20 21 7A	BLOCK2 JSR NEXTL
0130	7A88	CA	DEX
0131	7A89	D0 FA	BNE BLOCK2
0132	7A8B	A6 57	LDX TEMP2
0133	7A8D	A4 58	BLOCK3 LDY TEMP3
0134	7A8F	B1 54	BLOCK4 LDA (STARTL),Y
0135	7A91	49 80	EOR #\$80
0136	7A93	91 54	STA (STARTL),Y
0137	7A95	88	DEY
0138	7A96	D0 F7	BNE BLOCK4
0139	7A98	CA	DEX
0140	7A99	F0 06	BEQ BEND
0141	7A9B	20 21 7A	JSR NEXTL
0142	7A9E	4C 8D 7A	JMP BLOCK3
0143	7AA1	68	BEND PLA
0144	7AA2	AA	TAX
0145	7AA3	68	PLA
0146	7AA4	A8	TAY
0147	7AA5	68	PLA
0148	7AA6	60	RTS
0149	7AA7		.END

ERRORS = 0000

SYMBOL TABLE

SYMBOL	VALUE						
BEGIN	7A00	BEND	7AA1	BLOCK	7A7E	BLOCK1	7A83
BLOCK2	7A85	BLOCK3	7A8D	BLOCK4	7A8F	BORDER	7A33
HORIZ	7A00	NEXTH	7A04	NEXTL	7A21	NEXTV	7A14
STARTR	0055	STARTL	0054	TEMP1	0056	TEMP2	0057
TEMP3	0058	VERT	7A0E				

END OF ASSEMBLY

```

100 REM *****
101 REM *BASIC LOADER FOR MACHINE
102 REM *CODE PROGRAM DISPLAY.
109 REM *****
110 DATA31232
120 DATA98,48,A4,57,A5,58,91,54,88,D0,F9,68,A8,60
130 DATA98,48,A4,57,A2,00,A5,58,81,54,20,21,7A,88,10,F6,68,A8,60
140 DATA98,48,18,A9,28,65,54,85,54,A9,00,65,55,85,55,68,A8,60
150 DATA48,98,48,8A,48
160 DATAA9,A0,85,58,A9,FF,85,54,A9,7F,85,55,A9,28,85,57,20,00,7A
170 DATAA9,BF,85,54,A9,83,85,55,A9,28,85,57,20,00,7A
180 DATAA9,27,85,54,A9,80,85,55,A9,17,85,57,20,0E,7A
190 DATAA9,00,85,54,A9,80,85,55,A9,17,85,57,20,0E,7A
200 DATA68,AA,68,A8,68,60
210 DATA48,98,48,8A,48
220 DATAA6,56,20,21,7A,CA,D0,FA,A6,57
230 DATAA4,58,B1,54,49,80,91,54,88,D0,F7,CA,F0,06,20,21,7A
240 DATA4C,8D,7A,68,AA,68,A8,68,60
250 DATA*
260 READL
270 READA$
280 C=LEN(A$)
290 IFA$="*"THEN390
300 IFC<10RC>2THEN380
310 A=ASC(A$)-48
320 B=ASC(RIGHT$(A$,1))-48
330 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
340 IFN<00RN>255THEN380
350 POKEL,N
360 L=L+1
370 GOTO270
380 PRINT"BYTE"L=["A$"] "???"
390 END
READY.

```

HIGH DENSITY PLOTTING

One great drawback with having a display only 40 characters wide and 25 lines deep is the poor definition achievable when displaying data in graphical form. Although there is no way, short of modifying the circuitry, that the number of characters per line can be increased, one can improve the definition by clever manipulation of the graphics characters. Thus the five quarter square characters can be used to double the definition of a graph plotted on the screen. Similarly by using the seven characters with horizontal lines of different thickness one can draw a bar chart with a resolution of better than one in 160. The three programs in this section employ these techniques for graph plotting, barcharts and general graphical display.

DDPLOT

This subroutine draws a double density graph of a function defined as FNP(X) prior to the subroutine call. The axes are drawn across the centre of the screen and down the left hand side; no scale is drawn. Lines 24034 to 24040 perform the double density character selection from one of four quarter square characters.

Parameters used:

FNP(X) - the function to be plotted by the subroutine.

BARPLOT

BARPLOT draws a vertical barchart of up to 31 variables with a definition of 1 in 160. The variables are transferred to the subroutine as an integer array P%(X). The barchart is surrounded by a border and given a heading in reverse field characters in the centre of the top line of the screen. A vertical scale is given from zero to the maximum value to be displayed, the line increment is thus 1/160 of the maximum value. The horizontal scale numbers the bars from 1 to 31; this number was chosen to allow the bar chart to display daily data over a one month period and can be changed if desired. Lines 24166 to 24184 increment the vertical line in eight discrete steps, using different graphics characters for each increment.

Parameters used:

P\$ - variable for table heading; maximum length 40 characters.
P%(X)- table of data to be displayed in barchart; maximum 31 entries, all integers.
SP\$ - string variable of 40 space characters.

LPLPТСRC

This machine code program allows points to be plotted on the screen in double density format (ie. screen dimensions are 80 x 50 points). As with the subroutine DDPLLOT, this is done by using the quarter square graphics characters. This program however is much cleverer since it takes into account all the combinations of quarter square characters which can be placed in a single character space. The graphics character occupying a single character space will change when new points are plotted within that character space, without affecting existing plot points in the space. The program requires three variables which can be passed from a Basic program using POKE commands. These variables are the X coordinate passed as one byte, the Y coordinate passed as two bytes (the second byte is always set to zero). The third variable is set to 0 if the point is to be added to the display, and 1 if it is to be deleted. The program incorporates error checking and will set an error flag if an error condition is present. The error states are: firstly, plot coordinates are out of range, and secondly a plot point already exists at the coordinates specified. The program is located in the second cassette buffer and can be loaded either as a machine code program using the monitor, or with a Basic loader. This is contained in lines 10-400 of the program LINEPLOT.

Example: To plot a point at X - Y coordinates 14 - 25 use the following commands.

```
POKE 84,14    X co-ordinate
POKE 85,25    Y co-ordinate LSB
POKE 86,0     Y co-ordinate MSB
POKE 89,0     add point
```

To check the error status.

```
PEEK (998)
```

LINEPLOT

This program is an example of the use of the previous machine code program; lines 10-400 are a Basic loader for LLOTSRC. The program is designed to draw lines in quarter square characters between two sets of screen coordinates. These are input in line 1005. The remainder of the program calculates the position of the next point in the line, (parameters X1 and Y1), which are poked into the relevant locations prior to jumping to the machine code program in line 2270. The program could be easily modified to act as a subroutine within a larger program.

"

```

1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO DRAW GRAPH OF FUNCTION
1009 REM *****
1010 DEFFNP(X)=SIN(X/6.28)
1020 GOSUB24000
1030 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
24000 REM *****
24002 REM *SUBROUTINE TO DRAW A DOUBLE
24004 REM *DENSITY GRAPH OF THE FUNCTION
24006 REM *DEFINED BY - FNP(X)
24008 REM *****
24010 PRINT"J"
24012 FORX=32768TO33728STEP40:POKE X,101:NEXTX
24014 PRINT"XXXXXXXXXXXXXXXXXXXX"
24016 FORX=1TO79
24018 Y1=FNP(X)
24020 Y=24+24*Y1
24022 X2=INT(X/2):Y2=INT(Y/2)
24024 IFX2>39ORY2>25THENGOTO24044
24026 X1=X/2-X2:Y1=Y/2-Y2
24028 A=33728-Y2*40+X2
24030 IFX1<.5THENX1=0
24032 IFY1<.5THENY1=0
24034 IFX1=0ANDY1=0THENC=123:GOTO24042
24036 IFX1<>0ANDY1<>0THENC=124:GOTO24042
24038 IFX1<>0ANDY1=0THENC=108:GOTO24042
24040 IFX1=0ANDY1<>0THENC=126
24042 POKEA,C
24044 NEXTX:RETURN
READY.

```

```

100 DIMP%(31)
110 SP$="
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO DRAW A BARCHART.
1009 REM *****
1010 FORQ=1TO31
1020 INPUTP:IFP<0THEN 1050:REM **INPUT TILL NEGATIVE NUMBER**
1030 P%(Q)=P
1040 NEXTQ
1050 P$="EXAMPLE":REM **HEADING**
1060 GOSUB24100
1070 GETA$:IFA$=""THEN 1070:REM **WAIT TILL KEY PRESSED TO END**
1080 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
24100 REM *****
24102 REM *SUBROUTINE TO DRAW BARCHART
24104 REM *USING 31 VARIABLES STORED
24106 REM *AS P%(X). TABLE HEADING IS
24108 REM *TRANSFERED AS P$.
24110 REM *****
24112 A=LEN(P$):A1=(40-A)/2
24114 A1$=LEFT$(SP$,A1)
24116 PRINT" ";
24118 PRINTA1$:PRINT" "P$""
24120 PRINT" "
24122 FORX=32852TO33612STEP40:POKEX,101:NEXTX
24124 FORX=32885TO33685STEP40:POKEX,101:NEXTX
24126 PRINT"XXXXXXXXXXXXXXXXXXXXXXXXX 0 "
24128 PRINT" 1.....31 "
24130 B=0
24132 FORQ=1TO31
24134 A=P%(Q)
24136 IFA>BTHENB=A
24138 NEXTQ
24140 PRINT"XXXXX"B
24142 A=B/160
24144 FORQ=1TO31
24146 PRINT" "
24148 FORAV=1TO21:PRINT"X":NEXTAV
24150 PRINTTAB(Q+4);
24152 AS=INT(P%(Q)/A)
24154 AL=INT(AS/8)
24156 AF=AS-(8*AL)
24158 IFAL<1THENGOTO24166
24160 FORQS=0TOAL-1
24162 PRINT"X ";
24164 NEXTQS
24166 IFAF=0GOTO24186
24168 ONAFGOTO24170,24172,24174,24176,24178,24180,24182,24184
24170 PRINT"X":GOTO24186
24172 PRINT"X":GOTO24186
24174 PRINT"X":GOTO24186

```

24176 PRINT"■":GOTO24186
24178 PRINT"▣":GOTO24186
24180 PRINT"▤":GOTO24186
24182 PRINT"▥":GOTO24186
24184 PRINT"▦":GOTO24186
24186 NEXTQ
24188 RETURN
READY.

LINE#	LOC	CODE	LINE
0001	0000		*****
0002	0000		;*PET IN DOUBLE DENSITY FORMAT
0003	0000		;*X-COORD IN LOCATION 84
0004	0000		;*Y-COORD IN LOCATION 85
0005	0000		;*0 IN LOCATION 89 TO ADD
0006	0000		;*1 IN LOCATION 89 TO DELETE
0007	0000		;*ERROR FLAG IN LOCATION 998
0008	0000		;*1 OR 2 = PLOT OUT OF RANGE
0009	0000		;*4 = NON PLOTTABLE CHARACTER
0010	0000		;* ALREADY AT THESE COORDINATES
0011	0000		;* ON SCREEN
0012	0000		*****
0013	0000		;
0014	0000		;
0015	0000		XCOORD=\$54
0016	0000		YCOORD=\$55
0017	0000		AORD=\$59
0018	0000		BINDOFF=\$5A
0019	0000		REFRSH=\$E840
0020	0000		;
0021	0000		BEGIN=\$033A
0022	0000		*=BEGIN
0023	033A		;
0024	033A		;
0025	033A	A9 00	START LDA #\$0
0026	033C	8D E6 03	STA ERROR
0027	033F	85 5A	STA BINDOFF
0028	0341		;
0029	0341		TEST IF Y-COORD IS >49
0030	0341		;
0031	0341	A5 55	LDA YCOORD
0032	0343	C9 32	CMP #50
0033	0345	90 03	BCC YOK
0034	0347	EE E6 03	INC ERROR
0035	034A		;
0036	034A		TEST IF XCOORD IS >79
0037	034A		;
0038	034A	A5 54	YOK LDA XCOORD
0039	034C	C9 50	CMP #80
0040	034E	90 03	BCC XOK
0041	0350	EE E6 03	INC ERROR
0042	0353		;
0043	0353		RETURN IF OUT OF RANGE ERROR
0044	0353		;
0045	0353	2C E6 03	XOK BIT ERROR
0046	0356	F0 01	BEQ SORIG
0047	0358	60	RTS
0048	0359		;
0049	0359		INVERT SCREEN FROM TOP TO BOTTOM
0050	0359		(Y-COORDINATE)
0051	0359		;
0052	0359	A9 31	SORIG LDA #49
0053	035B	38	SEC
0054	035C	E5 55	SBC YCOORD
0055	035E	85 55	STA YCOORD

LINE#	LOC	CODE	LINE
0056	0360		;
0057	0360		;SAVE BOTTOM BIT OF X-COORD
0058	0360		;IN BINOFF
0059	0360		;
0060	0360	46 54	LSR XCOORD
0061	0362	26 5A	ROL BINOFF
0062	0364		;
0063	0364		;SAVE BOTTOM BIT OF Y-COORD
0064	0364		;IN BINOFF
0065	0364		;
0066	0364	46 55	LSR YCOORD
0067	0366	26 5A	ROL BINOFF
0068	0368		;
0069	0368		;MULTIPLY YCOORD BY 40 AND
0070	0368		;ADD SCREEN BASE ADDRESS
0071	0368		;
0072	0368	06 55	ASL YCOORD
0073	036A	06 55	ASL YCOORD
0074	036C	06 55	ASL YCOORD
0075	036E	A5 55	LDA YCOORD ;SAVE IN A-REG
0076	0370	06 55	ASL YCOORD
0077	0372	26 56	ROL YCOORD+1
0078	0374	06 55	ASL YCOORD
0079	0376	26 56	ROL YCOORD+1
0080	0378	18	CLC
0081	0379	65 55	ADC YCOORD
0082	037B	85 55	STA YCOORD
0083	037D	A5 56	LDA YCOORD+1
0084	037F	69 80	ADC #\$80 ;START OF SCREEN
0085	0381	85 56	STA YCOORD+1
0086	0383		;
0087	0383		;EXPAND BINOFF
0088	0383		;
0089	0383	A6 5A	LDX BINOFF
0090	0385	A9 01	LDA #1
0091	0387	85 5A	STA BINOFF
0092	0389	E0 00	EXP CPX #0
0093	038B	F0 05	BEQ ENDEXP
0094	038D	06 5A	ASL BINOFF
0095	038F	CA	DEX
0096	0390	90 F7	BCC EXP
0097	0392		;
0098	0392		;GET CHAR INTO A-REG
0099	0392		;
0100	0392	A4 54	ENDEXP LDY XCOORD
0101	0394	B1 55	LDA (YCOORD),Y
0102	0396		;
0103	0396		;CHECK CHAR IS VALID GRAPHIC
0104	0396		;
0105	0396	A2 00	LDX #0
0106	0398	DD CE 03	MOREC CMP TABLE,X
0107	039B	F0 0B	BEQ FOUND
0108	039D	E8	INX
0109	039E	E0 10	CPX #16
0110	03A0	90 F6	BCC MOREC

LINE#	LOC	CODE	LINE
0111	03A2		;
0112	03A2		;CHAR IS NOT IN TABLE
0113	03A2		;
0114	03A2	A9 04	LDA #4
0115	03A4	8D E6 03	STA ERROR
0116	03A7	60	RTS
0117	03A8		;
0118	03A8		;CHAR IS IN TABLE
0119	03A8		;SO ERASE OR ADD
0120	03A8		;
0121	03A8	A5 59	FOUND LDA AORD
0122	03AA	D0 07	BNE ERASPT
0123	03AC		;
0124	03AC		;ADD POINT TO SCREEN
0125	03AC		;
0126	03AC	8A	ADDPT TXA
0127	03AD	05 5A	ORA BINOFF
0128	03AF	18	CLC
0129	03B0	AA	TAX
0130	03B1	90 0A	BCC WAIT
0131	03B3		;
0132	03B3		;ERASE POINT FROM SCREEN
0133	03B3		;
0134	03B3	A5 5A	ERASPT LDA BINOFF
0135	03B5	49 FF	EOR #\$FF
0136	03B7	85 5A	STA BINOFF
0137	03B9	8A	TXA
0138	03BA	25 5A	AND BINOFF
0139	03BC	AA	TAX
0140	03BD		;
0141	03BD		;WAIT FOR SCREEN REFRESH
0142	03BD		;
0143	03BD	AD 40 E8	WAIT LDA REFRSH
0144	03C0	49 20	EOR #\$20
0145	03C2	29 20	AND #\$20
0146	03C4	F0 F7	BEQ WAIT
0147	03C6		;
0148	03C6		;WRITE NEW GRAPHIC CHAR TO SCREEN
0149	03C6		;
0150	03C6	BD CE 03	LDA TABLE,X
0151	03C9	A4 54	LDY XCOORD
0152	03CB	91 55	STA (YCOORD),Y
0153	03CD		;
0154	03CD		;RETURN SUCCESSFULLY
0155	03CD		;
0156	03CD	60	RTS
0157	03CE		;
0158	03CE		;TABLE OF GRAPHICS CHARACTERS
0159	03CE		;
0160	03CE	20	TABLE .BYTE \$20,\$7E,\$7B,\$61
0160	03CF	7E	
0160	03D0	7B	
0160	03D1	61	
0161	03D2	7C	.BYTE \$7C,\$E2,\$FF,\$EC
0161	03D3	E2	

LPLOTSRC.....PAGE 0004

LINE# LOC CODE LINE

```
0161 03D4 FF
0161 03D5 EC
0162 03D6 6C .BYTE $6C,$7F,$62,$FC
0162 03D7 7F
0162 03D8 62
0162 03D9 FC
0163 03DA E1 .BYTE $E1,$FB,$FE,$A0
0163 03DB FB
0163 03DC FE
0163 03DD A0
0164 03DE **+=8
0165 03E6 ERROR **+=1
0166 03E7 .END
```

ERRORS = 0000

SYMBOL TABLE

SYMBOL	VALUE						
ADDPT	03AC	AORD	0059	BEGIN	033A	BINOFF	005A
ENDEXP	0392	ERASPT	03B3	ERROR	03E6	EXP	0389
FOUND	03A8	MOREC	0398	REFRSH	E840	SORIG	0359
START	033A	TABLE	03CE	WAIT	03BD	XCOORD	0054
XOK	0353	YCOORD	0055	YOK	034A		

END OF ASSEMBLY

```

1 REM *****
2 REM *PROGRAM TO DRAW LINES ON SCREEN
3 REM *USING DOUBLE DENSITY MACHINE
4 REM *CODE PLOT SUBROUTINE.
5 REM *****
6 REM
7 REM
8 REM
9 REM **MACHINE CODE LOADER**
10 DATA826
20 DATAA9,00,8D,E6,03,85,5A
30 DATAA5,55,C9,32,90,03,EE,E6,03
40 DATAA5,54,C9,50,90,03,EE,E6,03
50 DATA2C,E6,03,F0,01,60
60 DATAA9,31,38,E5,55,85,55
70 DATA46,54,26,5A
80 DATA46,55,26,5A
90 DATA06,55,06,55,06,55,A5,55,06,55,26,56,06,55,26,56,18
91 DATA65,55,85,55,A5,56,69,80,85,56
100 DATAA6,5A,A9,01,85,5A,E0,00,F0,05,06,5A,CA,90,F7
110 DATAA4,54,B1,55,A2,00,DD,CE,03,F0,0B,E8,E0,10,90,F6
120 DATAA9,04,8D,E6,03,60
130 DATAA5,59,D0,07,8A,05,5A,18,AA,90,0A
140 DATAA5,5A,49,FF,85,5A,8A,25,5A,AA
150 DATARD,40,E8,49,20,29,20,F0,F7
160 DATABD,CE,03,A4,54,91,55,60
170 DATA20,7E,7B,61,7C,E2,FF,EC,6C,7F,62,FC,E1,FB,FE,A0
180 DATA*
200 READL
210 READA$
220 C=LEN(A$)
230 IFA$="*"THEN400
240 IFC<10RC>2THEN320
250 A=ASC(A$)-48
260 B=ASC(RIGHT$(A$,1))-48
270 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
280 IFN<00RN>255THEN320
290 POKEL,N
300 L=L+1
310 GOTO210
320 PRINT"BYTE"L="[A$] ???";
400 PRINT"Q";
600 REM
700 REM **ROUTINE TO DRAW LINES FROM START TO END COORDINATES**
800 REM
1000 PRINT"S";
1005 INPUTX1,Y1,X2,Y2
1010 GOSUB2000
1015 PRINT"S";
1020 GOTO1000
2000 REM
2010 REM **CHECK COORDINATES IN BOUND**
2020 REM
2030 IF(X1>=0ANDX1<=79)AND(X2>=0ANDX2<=79)THEN2060
2040 ER$="X OUT OF RANGE"
2050 RETURN
2060 IF(Y1>=0ANDY1<=49)AND(Y2>=0ANDY2<=49)THEN2090
2070 ER$="Y OUT OF RANGE"

```

```

2080 RETURN
2090 ER$=""
2100 XD=X2-X1
2110 YD=Y2-Y1
2120 REM **NEAREST DIAGONAL**
2130 A0=1:A1=1
2140 IFYD<0THENA0=-1
2150 IFXD<0THENA1=-1
2160 REM **NEAREST HORIZ/VERT**
2170 XE=ABS(XD):YE=ABS(YD):D1=XE-YE
2180 IFD1>0THEN2220
2190 S0=-1:S1=0:LG=YE:SH=XE
2200 IFYD>0THENS0=1
2210 GOTO2240
2220 S0=0:S1=-1:LG=XE:SH=YE
2230 IFXD>0THENS1=1
2240 REM **SET UP**
2250 TT=LG:TS=SH:UD=LG-SH:CT=SH-LG/2
2255 D=0
2260 REM **WHILE MORE POINTS DO**
2270 POKE84,X1:POKE85,Y1:POKE86,0:POKE89,D:SYS(826)
2280 IFCT>0THEN2320
2290 CT=CT+TS:X1=X1+S1:Y1=Y1+S0
2310 GOTO2360
2320 CT=CT-UD:X1=X1+A1:Y1=Y1+A0
2360 TT=TT-1
2370 IFTT<=0THENRETURN
2380 GOTO2270
READY.

```

GENERAL PURPOSE SCREEN HANDLER

This section deals with a general purpose subroutine capable of handling all data input and output from the screen (apart from graphical output). The subroutine incorporates full error checking, validation, and screen formatting. The basis of this subroutine is a table of nine arrays (in the subroutine these have 50 elements each, but this can be varied to suit the application and the machine). These arrays are used to store the parameters required by the subroutine, however, although parameters can be passed to and from this array from the main program, it is designed to be loaded with data from a disk file. The disk file will contain all the data required to print a message on the screen and input data according to certain parameters. Each element in the table contains nine variables (one from each array). These variables are: the message to be printed on the screen, its position on the screen, the start position of associated data input, input type and length. The last of the nine variables is an array where the data input is stored. They are as follows:

- LS - line of start of screen message
- CS - column of start of screen message
- LE - line of start of associated input
- CE - column of start of associated input
- P2\$ - input type
 - A = alphanumeric
 - N = numeric
 - D = date
 - Y = yes/no
 - C = numeric + check digit
- P3 - maximum input length
- P4 - minimum input length
- M\$ - message for display on screen
- PD\$ - data input by subroutine or data from main program to be output

These variables are stored on disk as a sequential file, (a parameter file), all associated elements being concatenated together as a single string variable 79 bytes long. This disk file is read by the subroutine starting at line 12000, which reads each of the concatenated strings (up to 50) in the data file, and dissects them into variables in their respective arrays (lines 12130-12170). The disk file is created when the program is written, using the program CRTMASK. The name given to this file will then be used as the variable PN\$ in the file read subroutine at line 12000. Before creating the file all the screen displays used by the program should be drawn out on graph paper as they will appear in the program (see diagram). Wherever possible the same input prompts should be reused in the same position on different screen displays, economising on the number of records in the file. This

subroutine also has a "help" facility which gives the operator additional information on what to do for a particular entry. The help facility is called by pressing the question mark key, a line of text explaining the current entry is then displayed on lines 24 and 25 of the screen. This line of text is also stored on the screen format parameter file, it is identical to a normal input message except that the parameters for input type, start and length, are set to zero. When all the data has been entered on to the file using the program CRTMASK the file should be terminated by entering a record consisting entirely of "Z"s, so that all 79 bytes of the record contain the character "Z". An example of the parameter file required by this subroutine is on the disk associated with this book it is called SCREEN1. . Having used the subroutine at line 12000 to load the data from the parameter file into the arrays, the main screen handling subroutine can now be run. This main subroutine which starts at line 28500 requires four parameters in both the input mode and output mode (besides the screen format parameter arrays). The most important of these is the input/output array PD\$; the pointer to which element in PD\$ is to be accessed is stored in the string PI\$. PI\$ is a string of concatenated two digit pointers into the screen format table, to screen format commands and messages required in a particular screen display. Thus a string for PI\$ of "09020501" will display messages stored in elements 9,2,5, and 1 of the screen format table and put the inputs in PD\$(9),PD\$(2),PD\$(5) and PD\$(1). A similar string PH\$ is used to point to the 'help messages' in the array. PH\$ is the same length as PI\$ and the 'help message' for a given input will be in the corresponding position to the pointers for the input in PI\$. With the PI\$ string "09020501" a corresponding string for PH\$ could be "04000600" where 'help message' 4 will be displayed if requested during input 9 and help 6 during input 5. An input of 00 in PH\$ means no 'help message', thus input 2 will have no 'help message'. The last parameter required is PO. This is used to indicate whether the subroutine is to be in the input or output mode. If PO is set to 0 then the subroutine is in the normal input mode, and the result of each input line will be put into the corresponding element of PD\$. If PO is 1 then the subroutine will be in the output mode and the contents of the corresponding elements of PD\$ will be displayed after the message starting at the input start location. Lines 2000-2040 are an example of parameter calling routine using the screen handling subroutine and setting up the parameters for input of data. Lines 3000-3050 are simply a means of displaying the contents of the output parameters in array PD\$ which have been input by the subroutine.

All the standard data input command keys are used (see section on data input subroutines p.13). Note that when a field is aborted the word "abort" is placed in the corresponding element of array PD\$.

Parameters used:

Arrays:

LS(X) - line of start of message,two bytes long
CS(X) - column of start of message,two bytes long
LE(X) - line of start of input or output variable,two bytes long
CE(X) - column of start of input or output variable,two bytes long
P2\$(X)- input type, one byte long
P3(X) - maximum input length, two bytes long
P4(X) - minimum input length, two bytes long
M\$(X) - screen input message or help message max 66 bytes long
PD\$(X)- result strings either from input or to be output, maximum length defined by contents of variable P3(X)

Mask strings:

PI\$ - string of pointers P1 into arrays M\$,LS,CS,LE,CE,P2\$,P3,P4 and PD\$; each pointer occupies two bytes and there is no limit on the number of pointers in the string.
PH\$ - string of pointers into array M\$ used for help messages, each pointer occupies two bytes and corresponds to a pointer in the same position in PI\$. A 00 entry signifies that no help message is available for that entry.

Various:

PO - set to one for subroutine to function in output mode and set to zero for normal input mode.
PN\$ - screen format file name required when screen parameter file is read during program initialisation, should contain the file name then ",S,R"(eg PN\$="SCREEN1,S,R" where SCREEN1 is the file name)
SP\$ - string of 40 space characters

Data file structure of file SCREEN1

Lines

[illegible]

```

100 SP$="
110 CR$=CHR$(13)
1000 REM *****
1001 REM *THIS PROGRAM IS DESIGNED TO
1002 REM *CREATE A DISPLAY DATA FILE
1003 REM *FOR USE BY THE PROGRAM
1004 REM *CRTFORMAT.
1009 REM *****
2000 INPUT "ENTER CRT MASK FILE NAME";A$
2050 B$="@1:" + A$ + ", SEQ, WRITE"
2060 OPEN 15, 8, 15: GOSUB 30000
2100 OPEN 2, 8, 2, B$: GOSUB 30000
2105 PRINT#2, A$CR$
2110 FOR Q=0 TO 50: REM **MAX NUMBER OF RECORDS IN FILE**
2120 INPUT "LINE OF START OF MESSAGE";B$
2130 B$=RIGHT$("00"+B$,2)
2140 CU$=B$
2150 INPUT "COLUMN OF START OF MESSAGE";B$
2160 B$=RIGHT$("00"+B$,2)
2170 CU$=CU$+B$
2180 INPUT "LINE OF START OF INPUT";B$
2190 B$=RIGHT$("00"+B$,2)
2200 CU$=CU$+B$
2210 INPUT "COLUMN OF START OF INPUT";B$
2220 B$=RIGHT$("00"+B$,2)
2230 CU$=CU$+B$
2240 INPUT "VARIABLE TYPE- A, N OR M ";B$
2250 B$=RIGHT$(B$,1)
2260 CU$=CU$+B$
2270 INPUT "MAXIMUM INPUT LENGTH";B$
2280 B$=RIGHT$("00"+B$,2)
2290 CU$=CU$+B$
2300 INPUT "MINIMUM INPUT LENGTH";B$
2310 B$=RIGHT$("00"+B$,2)
2320 CU$=CU$+B$
2330 INPUT "MESSAGE ";M$
2340 IF LEN(M$)>66 THEN 2330
2350 M$=LEFT$(M$+SP$,66)
2420 IF LEFT$(CU$,2)="ZZ" THEN Q=50
2430 REM **PRINT CONCATINATED RECORD TO DISK**
2440 PRINT#2, M$+CU$CR$
2470 NEXT Q
2500 CLOSE 2: END
30000 REM **DISK ERROR DETECTION**
30000 INPUT#15, EN$, EM$, ET$, ES$
30010 IF EN$="00" THEN RETURN
30020 PRINT EM$, EN$, ET$, ES$
30030 CLOSE 2: END
READY.

```

```

100 SP$=""
110 DIMP$(50),LS(50),CS(50),LE(50),CE(50),P2$(50),P3(50),P4(50),M$(50)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL -
1002 REM *FIRST TO INPUT SCREEN FORMAT
1003 REM *FILE FROM DISK THEN RUN SCREEN
1004 REM *HANDLER PROGRAM TO DISPLAY
1005 REM *SAMPLE SCREEN OUTPUT WITH
1006 REM *DIFFERENT TYPES OF INPUT AND
1007 REM *HELP MESSAGES.
1009 REM *****
1100 REM
1200 REM
1300 REM
1400 PN$="SCREEN1"
1410 GOSUB12000
1500 REM
1600 REM
1700 REM
2000 P1$="0102030405"
2010 PH$="0600070800"
2015 PRINT"J";
2016 PO=0
2020 GOSUB28500
2030 END
4000 REM
5000 REM
6000 REM
7000 REM
8000 REM
12000 REM *****
12010 REM *S/R TO INPUT SCREEN FORMAT
12020 REM *DATA FROM FILE NAME PN$
12030 REM *ALSO INITIALISE ARRAYS USED
12040 REM *BY SCREEN HANDLER.
12050 REM *****
12060 FORQ=0TO50
12070 M$(Q)=" ":PD$(Q)=" "
12080 NEXTQ
12090 OPEN2/8,2,PN$
12100 FORQ=0TO50
12110 INPUT#2,A$
12120 IFMID$(A$,67,2)="ZZ"THENQ=50
12130 LS(Q)=VAL(MID$(A$,68,2)):CS(Q)=VAL(MID$(A$,70,2))
12140 LE(Q)=VAL(MID$(A$,72,2)):CE(Q)=VAL(MID$(A$,74,2))
12150 P2$(Q)=MID$(A$,76,1):P3(Q)=VAL(MID$(A$,77,2))
12160 P4(Q)=VAL(MID$(A$,79,2))
12170 M$(Q)=LEFT$(A$,66)
12300 NEXTQ
12310 CLOSE2
12320 RETURN
20000 REM
21000 REM
22000 REM
23000 REM
25000 REM *****
25010 REM *CHECK DIGIT CALCULATION

```

```

25020 REM *NUMBER INPUT AS P$ AND
25030 REM *CHECK DIGIT OUTPUT AS PC
25040 REM *****
25050 A=VAL(P$):BD=10
25060 IF LEFT$(P$,1)="-" THEN A=VAL(RIGHT$(P$,LEN(P$)-1)):BD=50
25070 A1=0:A2=2
25080 B=INT(A/10)
25090 A1=A1+(A-(B*10))*A2
25100 IF B=0 THEN 25140
25110 A=B
25120 A2=A2+1
25130 GOTO 25080
25140 AD=A1-(INT(A1/11)*11)
25150 AD=11-AD
25160 PC=INT(AD)+BD:RETURN
26000 REM
26500 REM
27000 REM
27500 REM
28500 REM *****
28510 REM *START OF SCREEN FORMAT S/R
28520 REM *****
28530 I=LEN(P1$)
28540 FOR AI=1 TO I STEP 2
28550 IF AI<1 THEN AI=1
28560 P1=VAL(MID$(P1$,AI,2))
28570 PH=VAL(MID$(PH$,AI,2))
28580 PD$(P1)="*"
28670 REM *****
28680 REM *CLR HELP & PRINT MESSAGE
28690 REM *****
28700 IF PL=2 THEN COL=1:LNE=24:GOSUB 29800:PRINT SP$;
28705 PL=0
28710 COL=CS(P1):LNE=LS(P1):GOSUB 29800:PRINT LEFT$(M$(P1),40);
28720 COL=CE(P1):LNE=LE(P1):GOSUB 29800
28730 IF PO=1 THEN PRINT PD$(P1):GOTO 29410
28740 REM ***CLEAR INPUT VARIABLES***
28750 P$=""
28760 REM ***PRINT INPUT PROMPT***
28770 FOR Q=1 TO P3(P1):PRINT "_";:NEXT Q
28780 FOR Q=1 TO P3(P1):PRINT "||";:NEXT Q
28790 REM ***GET INPUT CHARACTER***
28800 REM ***& TEST FOR COMMANDS***
28810 POKE 158,0
28815 GET A$:IFA$="" THEN 28815
28820 IFA$=CHR$(13) THEN 28960
28830 IFA$="←" THEN GOTO 29010
28840 IFA$="↑" THEN PL=1:GOTO 29390
28850 IFA$="!" THEN GOTO 29400
28860 IFA$="?" THEN PL=2:GOTO 29760
28870 REM
28880 REM ***DELETE LINE***
28890 REM
28900 IFA$<>CHR$(20) THEN 29080
28905 IF P$="" THEN 28810
28910 PL=LEN(P$):FOR Q=1 TO PL:PRINT "||";:NEXT
28920 GOTO 28750
28930 REM
28940 REM ***CHECK FOR RETURN***
28950 REM

```

```

28960 IFLEN(P$)>=P4(P1)THEN29360
28970 GOTO28810
28980 REM
28990 REM ***DELETE CHARACTER***
29000 REM
29010 IFP$=""THEN28810
29030 P$=LEFT$(P$,LEN(P$)-1)
29040 A$="|||":GOTO29300
29050 REM
29060 REM ***DETERMINE INPUT TYPE***
29070 REM
29080 IFP2$(P1)="A"THEN29240
29090 IFP2$(P1)="N"ORP2$(P1)="C"ORP2$(P1)="D"THEN29170
29100 IFP2$(P1)="Y"THEN29550
29110 REM
29120 REM
29130 GOTO29240
29140 REM
29150 REM ***NUMERIC INPUT ***
29160 REM
29170 IFA$="."THEN29290
29180 IFA$="-"ANDP$=""THEN29290
29190 IFA$<"0"ORA$>"9"THEN28810
29200 GOTO29290
29210 REM
29220 REM ***ALPHANUMERIC INPUT***
29230 REM
29240 IFA$<" "THEN28810
29250 IFA$>"Z"ANDA$<"|"THEN28810
29260 REM
29270 REM ***CONCATINATE***
29280 REM
29290 P$=P$+A$
29300 PRINTA$;
29310 IFLEN(P$)<P3(P1)THEN28810
29320 IFLEN(P$)=P3(P1)ANDP2$(P1)="C"THEN29450
29330 IFLEN(P$)=P3(P1)ANDP2$(P1)="D"THEN29630
29340 REM
29350 REM ***PLACE IN INPUT ARRAY***
29360 P1$(P1)=P$
29370 REM
29380 REM ***CHECK FOR UP ONE LINE***
29390 IFPL=1THENAI=AI-4
29400 NEXTAI
29410 RETURN:REM ***S/R EXIT***
29420 REM
29430 REM ***CHECK DIGIT INPUT
29440 REM
29450 PRINT"|||":GOSUB25000
29460 GETA$: IFA$=""THEN29460
29470 GETA1$: IFA1$=""THEN29470
29480 A$=A$+A1$:PRINTA$;:AC=VAL(A$)
29490 IFAC=PCTHEN29360
29500 FORQ=1TO100:NEXT
29510 PRINT"||| |||":GOTO29450
29520 REM
29530 REM ***YES/NO INPUT***
29540 REM
29550 IFA$="Y"THENP$="1":PRINT"  YES":GOTO29360
29560 IFA$="N"THENP$="0":PRINT"  NO ":GOTO29360

```

```

29570 GOT028810
29580 REM
29590 REM
29600 REM *****
29610 REM *DATE INPUT AND VALIDATION
29620 REM *****
29630 AM$="312831303130313130313031"
29640 AL$="312931303130313130313031"
29650 AD=VAL(LEFT$(P$,2))
29660 AM=VAL(MID$(P$,3,2))
29670 AY=VAL(RIGHT$(P$,2))
29680 IFAM<10RAM>12ORAY<70ORAY>85ORAD<010RAD>31THEN29720
29690 AR=1900+AY:IFINT(AR/4)=AR/4THENAM$=AL$
29700 IFAD>VAL(MID$(AM$,AM*2-1,2))THEN29720
29710 GOT029360
29720 GOT028910
29730 REM
29740 REM ***HELP MESSAGE PRINT***
29750 REM
29760 IFPH=0THEN28710
29770 COL=1:LNE=24:GOSUB29800
29780 PRINT"  HELP -  ";LEFT$(M$(PH),66);
29790 GOT028710
29800 REM *****
29810 REM *CURSOR CONTROL SUBROUTINE
29820 REM *****
29830 AC$="!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
29840 AD$="!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
29860 PRINT"  ";
29870 IFCOL>1THENPRINTLEFT$(AC$,COL-1);
29880 IFLNE>1THENPRINTLEFT$(AD$,LNE-1);
29910 RETURN
READY.

```

ARRAY SORTS

Where data is stored in an array, it is frequently desirable to have the data in an alphabetic or numerical order. The principle reason for storing data in an ordered format is that it is quicker and easier to find an item in an ordered array than in an unordered array. The problem is that data is rarely entered into an array in an ordered form. To order the array a special sort subroutine is used to convert the unordered data into ordered form. There is a wide range of different sort algorithms the choice of which one to use depends on several factors, such as the time taken to sort an array of a certain size and the amount of extra memory required for temporary storage by the sort program. The general principle being the quicker the sort the larger the amount of memory required to store the variables. The time taken to sort an array depends on the the array size. The relationship between size and speed is exponential. Doubling the array size will require much longer than double the time to sort. The following are three different types of simple array sort subroutines.

BUBBLESORT

This is the simplest of all sort algorithms and also one of the slowest, however, as it uses virtually no temporary storage, a bubblesort is the ideal choice if memory is at a premium and sort time is not important. The first version of bubblesort will sort an array PSS\$ of PN elements into alphabetical order. The second version is identical except that the sort key length PK can be selected. Meaning that if the sort key is set to 3, the array will be sorted so that only the first three characters of each element are in order, the order of any subsequent characters is ignored. Note: alphabetic sorts will also sort numbers providing they are all integers and stored in string format. Fractional numbers are likely to be sorted incorrectly.

Parameters used

PSS\$ - array to be sorted, sorted output stored in same array
PN - number of elements in array
PK - length of sort key

SHELLMETZNER

A Shell-Metzner sort is considerably faster than a

bubblesort, but as it is also very economical on temporary storage requirements, it is probably the best simple array sort algorithm for general purpose use. The first version of the Shellmetzner sort will sort an array PSS of PN elements into alphabetic order with a sort key length of PK. The second version will sort PN numeric variables, stored in string format in array PSS, into numeric order, placing fractional numbers into their proper positions as well as integers.

Parameters used:

PSS - array to be sorted; sorted output put in same array
PN - number of elements in array PSS
PK - sort key length

REPLACESORT

This is the fastest of the three algorithms in this section, but unfortunately it requires a considerable amount of extra memory. The main reason for the extra memory requirement is that the output array is different from the input array, also an index array of twice the number of elements in the input array is required. The output array is not absolutely necessary, since the index array will contain information about the proper order of the data in the input array. Line 26270 could therefore be omitted to save space occupied by the output array. If this line is omitted then the index array must be used to indicate the order of the input array. The existence of an index array is useful when sorting a table consisting of more than one array. By sorting one array the pointers in the index array can be used with the other arrays, thereby keeping associated elements in different arrays together.

Parameters used:

PN - number of elements in input and output arrays
(index array has 2 x PN elements)
PU\$(PN)- input array of PN elements
PSS(PN)- output array of PN elements
PI(2PN)- index array of 2xPN elements

```

100 DIM PS$(10)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL
1002 REM *TO SORT DATA IN ARRAY PS$()
1003 REM *USING A BUBBLE SORT.
1009 REM *****
1010 FORQ=1TO10
1020 INPUTPS$(Q):REM **INPUT DATA INTO PS$()**
1030 NEXT
1040 PN=10
1050 GOSUB20000 :REM **SORT**
1060 FORQ=1TO10
1070 PRINTPS$(Q):REM **PRINT SORTED ARRAY**
1080 NEXTQ
1090 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
20000 REM*****
20010 REM*BUBBLE SORT OF ARRAY PS$
20020 REM*NUMBER OF ELEMENTS IN
20030 REM*ARRAY PN.
20040 REM*****
20050 FORQ=1TOPN-1
20060 FORX=(Q+1)TOPN
20070 IFPS$(X)>=PS$(Q)THEN20110
20080 A$=PS$(Q)
20090 PS$(Q)=PS$(X)
20100 PS$(X)=A$
20110 NEXTX
20120 NEXTQ
20130 RETURN
READY.

```

```

23000 REM*****
23010 REM*BUBBLE SORT OF ARRAY PS$
23020 REM*NUMBER OF ELEMENTS IN
23030 REM*ARRAY PN KEY LENGTH PK
23040 REM*****
23050 FORQ=1TOPN-1
23060 FORX=(Q+1)TOPN
23070 IFLEFT$(PS$(X),PK)>=LEFT$(PS$(Q),PK)THEN23110
23080 A$=PS$(Q)
23090 PS$(Q)=PS$(X)
23100 PS$(X)=A$
23110 NEXTX
23120 NEXTQ
23130 RETURN
READY.

```

```

100 DIM PS$(10)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL -
1002 REM *TO INPUT AND SORT AN ARRAY PS$()
1003 REM *USING A SHELL METZNER SORT.
1009 REM *****
1010 FORQ=1TO10
1020 INPUT PS$(Q)
1030 NEXTQ
1040 INPUT "SORT KEY LENGTH":PK
1050 PN=10
1060 GOSUB23200
1070 FORQ=1TO10
1080 PRINTPS$(Q)
1090 NEXTQ
1100 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
23200 REM*****
23210 REM*SHELL-METZNER SORT OF ARRAY
23220 REM*PS$, NUMBER OF ELEMENTS PN
23230 REM*KEY LENGTH PK.
23240 REM*****
23250 AN=PN:AM=PN
23260 AM=INT(AM/2):IFAM=0THEN RETURN
23270 AJ=1:AK=AN-AM
23280 AI=AJ
23290 AL=AI+AM
23300 IFLEFT$(PS$(AI),PK)<=LEFT$(PS$(AL),PK)THEN23370
23310 A$=PS$(AI)
23320 PS$(AI)=PS$(AL)
23330 PS$(AL)=A$
23340 AI=AI-AM
23350 IFAI<1THEN23370
23360 GOTO23290
23370 AJ=AJ+1
23380 IFAJ>AKTHEN23260
23390 GOTO23280
READY.

```

```

100 DIM PS$(10)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL -
1002 REM *TO INPUT AND SORT AN ARRAY
1003 REM *OF NUMERIC VARIABLES PS$( )
1004 REM *USING A SHELL METZNER SORT.
1009 REM *****
1010 FORQ=1TO10
1020 INPUT PS$(Q): REM**INPUT ARRAY**
1030 NEXTQ
1040 PN=10
1050 GOSUB23200: REM**SORT**
1060 FORQ=1TO10
1070 PRINTPS$(Q): REM**PRINT SORTED ARRAY**
1080 NEXTQ
1090 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
23200 REM*****
23210 REM*SHELL-METZNER SORT OF ARRAY
23220 REM*PS$, NUMBER OF ELEMENTS PN
23230 REM*(NUMERIC VARIABLES)
23240 REM*****
23250 AN=PN:AM=PN
23260 AM=INT(AM/2):IFAM=0THEN RETURN
23270 AJ=1:AK=AN-AM
23280 AI=AJ
23290 AL=AI+AM
23300 IFVAL(PS$(AI))<=VAL(PS$(AL))THEN23370
23310 A$=PS$(AI)
23320 PS$(AI)=PS$(AL)
23330 PS$(AL)=A$
23340 AI=AI-AM
23350 IFAI<1THEN23370
23360 GOTO23290
23370 AJ=AJ+1
23380 IFAJ>AKTHEN23260
23390 GOTO23280
READY.

```

```

100 DIM PI(20),PU$(10),PS$(10)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL -
1002 REM *TO SORT AN ARRAY PS$() USING
1003 REM *A REPLACESORT. THE CONTENTS OF
1004 REM *BOTH THE UNSORTED, INDEX, AND
1005 REM *SORTED ARRAYS ARE PRINTED AFTER
1006 REM *THE SORT.
1009 REM *****
1010 FORQ=1TO10
1020 INPUTPU$(Q)
1030 NEXTQ
1040 PN=11
1050 GOSUB26200
1060 FORQ=1TO10
1070 PRINTPU$(Q),PS$(Q),PI(Q)
1080 NEXTQ
1100 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
26200 REM *****
26201 REM *REPLACEMENT SORT, UNSORTED
26202 REM *DATA IN PU$(I), SORTED IN
26203 REM *PS$(I). NUMBER OF ENTRIES
26204 REM *IN ARRAY PN.
26209 REM *****
26210 B=0:A=PN-1:FORQ=0TOA:PI(Q)=Q:NEXTQ
26220 FORQ=0TOPN*2-3STEP2
26230 A=A+1:A1=PI(Q):A2=PI(Q+1)
26240 GOSUB26350
26250 PI(A)=PI:NEXTQ
26260 A4=A4-1:A3=PI(A):IFA3<0THEN RETURN
26270 PS$(B)=PU$(A3):B=B+1
26280 PI(A3)=A4
26290 A3%=A3/2:Q=A3%*2:A3=PN+A3%:IFA3>0GOTO26260
26300 A1=PI(Q):A2=PI(Q+1)
26310 IFA1<0THENPI=A2:GOTO26340
26320 IFA2<0THENPI=A1:GOTO26340
26330 GOSUB26350
26340 PI(A3)=PI:GOTO26290
26350 PI=A1:IFPU$(A2)<PU$(A1)THENPI=A2
26360 RETURN
READY.

```

SORTING LARGE FILES

Simple sorts require that the data is held in internal memory, usually as an array. This is feasible with small quantities of data, but with large files, stored on disk or tape, it may be impossible because of insufficient storage space. The solution is to use a technique utilising two data files, including the one holding the source file. One technique involves splitting the data on the input file (file A) into small blocks, which are then sorted and output to the output file (file B). The output file contains small sorted sets of data, if the output file is sorted a second time the result will be no different to the first. The reason for this being that the work area (the number of elements sorted at a time) is of a fixed size, this is overcome by sorting the data using work areas of different sizes. Thus when data is being sorted from file A to file B the work area may be set to the maximum size of 100 elements, but when data is sorted from B to A the work area is varied between 61 and 69 elements. The work area is varied to avoid the possibility of the program getting in a loop which cannot be broken. If the work area is 100 and 60 then any file with more than 600 records will be sorted up to the 600th record, then another sorted sequence would start from record 601. Since any sort program will stop only when the whole file is in sorted order, the program will in this case run indefinitely. The second technique also involves the use of two files, one is the master data file and the second is a short transaction file. Data is never input directly to the master file, it is instead input to the transaction file. Since the transaction file is short, probably no more than 100 records, it can be easily sorted in memory. The sorted transaction file is then merged with the much larger master file, each record in the transaction file being put in its correct position. This technique of sorting a transaction file, then merging it with the master file, is much faster than sorting the entire master file.

FILESORT

This is not a subroutine, but a complete program which can be used as a file sort utility within a suite of programs. It can be used to sort a complete master file, or to sort a transaction file (the purpose for which it was designed). The input file which in the program is given the name "UNSORTED" is on disk drive 1, after the sort procedure this file is replaced by the output file "SORTED" also on drive 1. Variables initialised at the beginning of the program allow one to select the sort key start position and the sort key length, where the sort key is that part of the record which is to be sorted. Variables also set up at the beginning of

the program, determine the maximum number of records which can be sorted in core. These values depend on the size of machine and the length of each record, and must be calculated for each application. The sequential disk file to be sorted will consist of an indeterminate number of fixed length records (where each record is a string of less than 80 bytes). The last record on the file will be a terminator record containing a string of "Z"s at least four characters long. The time taken to sort a file is obviously dependent on the number of records in the file, it is preferable to keep transaction files fairly small.

Since this program is designed to be part of a program suite, line 1990 which terminates the program incorporates the commands to chain the program back to a menu program (or the next program in the suite). The poke values should be set to the values of locations 42 and 43 when the menu program is loaded (these two locations show the amount of memory occupied by a program), the program chained from the sort is here called "MENU". For further explanation on program chaining see section on menus (p.129).

Parameters used:

MX - number of records which can be sorted in core
A\$(MX+2)- sort table A
B\$(MX) - sort table B
SKSP - sort key start position
KLN - sort key length
IPFILE\$ - input file name
OPFILE\$ - output file name

FILEMERGE

This program is intended for use with a sorted transaction file ("SORT") produced by the file sort program "Filesort". The transaction file is merged with the master file ("MASTER") to create a new master file. The program requires a temporary copy of the master file (called "DUMMY"), so sufficient disk space should be available for this. Like the transaction file the master file has an indeterminate number of records, and is terminated by a record consisting of a string of four "Z"s. During the merge the number of records on both the transaction file and the master file are displayed on the screen. Full error checking is incorporated, if there is any discrepancy the merge may be aborted prior to deletion of the original files. As with Filesort this program is intended to be part of a suite of programs each chained to a menu; line 1150 performs this chaining back to the program "MENU".

Parameters used:

No input parameters are required by this program.

NOTE: since no records are directly input to the master file, the file must be created when the system is first set up. This requires a single terminator record ("ZZZZ") to be written to the file. The system can now be run and further records will be automatically added to the master file.

```

1000 REM *****
1010 REM *ON DISK VARIABLES "MX" MUST BE
1020 REM *SET TO MAX. NO. OF RECORDS
1030 REM *THAT CAN BE SORTED IN CORE.
1040 REM *TABLES "A$" & "B$" MUST BE
1050 REM *DIMENSIONED AS "MX+2" & "MX".
1060 REM *IF MORE THAN "MX" RECORDS ARE
1070 REM *TO BE SORTED THEY WILL BE SORT
1080 REM *-ED IN BATCHES OF "MX" RECORDS
1090 REM *& EACH SORTED STRING WILL BE
1100 REM *COPIED TO A WORK FILE ON DISK.
1110 REM *TWO WORK FILES ARE USED FOR
1120 REM *OUTPUT & ARE USED ALTERNATELY.
1130 REM *WHEN ALL RECORDS HAVE BEEN
1140 REM *SORTED THE STRINGS ON THE WORK
1150 REM *FILES ARE MERGED TO FORM THE
1160 REM *SORTED OUTPUT FILE.
1170 REM *****
1180 DIM A$(256), S(10), B$(254): Z4$="ZZZZ": MX=254: FT$="Y"
1250 REM *****
1260 REM *SORT CONTROL DATA
1270 REM *SI/O FILE NAMES & START
1280 REM *& LENGTH OF SORT KEY
1290 REM *****
1300 OPEN I0, 8, 15
1310 SKSP=1
1320 KLN=16: IP$="UNSORTED": OP$="SORTED"
1350 REM *****
1360 REM * OPEN INPUT FILE & STORE
1370 REM *RECORDS IN "B" & KEYS+TAGS
1380 REM *IN "A". WHEN TABLE FULL OR
1390 REM *END OF FILE SORT TABLE "A"
1400 REM *****
1410 N=0: FL$="1:" + IP$ + ", S, R": OPEN 2, 8, 2, FL$: IF ST>0 THEN GOSUB 2630
1420 INPUT #2, IR$
1430 IF LEFT$(IR$, 4) = "ZZZZ" THEN CLOSE 2: GOTO 1510
1440 N=N+1: A$(N+1) = MID$(IR$, SK, KL) + RIGHT$( " " + STR$(N), 3): B$(N) = IR$
1450 IF N < MX THEN 1420
1460 REM *****
1470 REM * SET UPPER & LOWER BOUNDS
1480 REM * SORT PARAMS. ENTER SORT
1490 REM * AT LINE 4000
1500 REM *****
1510 A$(1) = " " : A$(N+2) = "ZZZZZZZZZZZZ"
1520 FOR I=1 TO 10: S(I) = 0: NEXT
1530 P=1: M=10: I=N: R=N+1
1540 IF N=0 AND S0=1 THEN S0=2: B$(1) = Z4: A$(2) = "ZZ001": N=1: GOTO 1660
1550 IF N=0 THEN 1820
1560 REM *****
1570 REM *CHECK IF ANY MORE RECORDS TO
1580 REM *BE SORTED. IF YES THEN COPY
1590 REM *TABLE "B" IN SEQUENCE TO
1600 REM *FILE "ASORT" OR "BSORT". IF
1610 REM *EOF THEN FORM OUTPUT FILE
1620 REM *FROM TABLE "B" IF SORT USED
1630 REM *ONCE ELSE MERGE "B" WITH
1640 REM *FILES "ASORT" & "BSORT"

```

```

1650 REM*****
1660 IFS0=1ANDLEFT$(IR$,4)=Z4$THENLF=14:SA=14:FL$=OP$+"",S,W":GOTO1710
1670 LF=3:IFS0T/2=INT(S0T/2)THENLF=4
1680 SA=LF:IFS0T>2THEN1730
1690 F$="A":IFS0T=2THENF$="B"
1700 FL$=F$+"SORTFL,SEQ,WRITE"
1710 OPENLF,8,SA,"@1:"+FL$
1720 IFST>0THENGOSUB2630
1730 FORI=1TON
1740 PRINT#LF,B$(VAL(RIGHT$(A$(I+1),3)));CHR$(13);:NEXT
1750 REM*****
1760 REM*IF SORT=1 & EOF THEN END. IF
1770 REM*NOT EOF THEN SORT NEXT SET
1780 REM* OF RECORDS ELSE MERGE FILES
1790 REM*****
1800 IFLEFT$(IR$,4)<>Z4$THENN=0:GOTO1420
1810 IFLF=14THEN1920
1820 PRINT#3,Z4$:CHR$(13);:IFST>0THENGOSUB2630
1830 CLOSE3:IFST>0THENGOSUB2630
1840 PRINT#4,Z4$:CHR$(13);:IFST>0THENGOSUB2630
1850 CLOSE4:IFST>0THENGOSUB2630
1860 REM*****
1870 REM* ENTER MERGE S/R TO FORM
1880 REM* OUTPUT FILE
1890 REM*****
1900 GOSUB2740
1910 GOTO1970
1920 REM*****
1930 REM*ALL RECORDS ON OUTPUT FILE
1940 REM*****
1950 PRINT#14,Z4$:CHR$(13);:IFST>0THENGOSUB2630
1960 CLOSE14:IFST>0THENGOSUB2630
1970 PRINT"*****END OF SORT"
1980 PRINT"*****NO. OF RECORDS SORTED = ";RECS
1990 POKE042,10:POKE043,36:CLR:LOAD"MENU",8
2000 P=1:M=10
2010 PRINT"*";
2020 L=2
2030 R=N+1
2040 IFR=L<MTHEN2410
2050 I=L
2060 J=R
2070 K=A$(L)
2080 IFK$>A$(J)THEN2110
2090 J=J-1
2100 GOTO2080
2110 IFJ>ITHEN2140
2120 A$(I)=K$
2130 GOTO2270
2140 A$(I)=A$(J)
2150 I=I+1
2160 IFA$(I)>=K$THEN2190
2170 I=I+1
2180 GOTO2160
2190 REM
2200 IFJ>ITHEN2240
2210 A$(J)=K$
2220 I=J
2230 GOTO2270
2240 A$(J)=A$(I)

```

```

2250 J=J-1
2260 GOT02080
2270 IFR-I<I-LTHEN2340
2280 S(P)=R
2290 P=P+1
2300 S(P)=I+1
2310 P=P+1
2320 R=I-1
2330 GOT02040
2340 S(P)=I-1
2350 P=P+1
2360 S(P)=L
2370 P=P+1
2380 L=I+1
2390 GOT02040
2400 REM STRAIGHT INSERT
2410 J=L
2420 J=J+1
2430 IFJ>RTHEN2520
2440 K$=A$(J)
2450 I=J-1
2460 IFA$(I)<=K$THEN2500
2470 A$(I+1)=A$(I)
2480 I=I-1
2490 GOT02460
2500 A$(I+1)=K$
2510 GOT02420
2520 IFP=1THEN2620
2530 P=P-1
2540 L=S(P)
2550 P=P-1
2560 R=S(P)
2570 GOT02040
2580 REM*****
2590 REM*TABLE "A" SORTED RETURN TO
2600 REM*MAIN PATH OF PROGRAM
2610 REM*****
2620 RETURN
2630 INPUT#10,R1$,R2$,R3$,R4$
2640 IFVAL(R1$)=0THENRETURN
2650 PRINT"DISC ERROR"
2660 PRINT"ERROR NO ";R1$
2670 PRINT"ERROR MSG: ";R2$
2680 PRINT"PARAMETER 1 ";R3$
2690 PRINT"PARAMETER 2 ";R4$
2700 PRINT""
2710 INPUT"CONTINUE ? (Y/N) ";YN$:IFYN$="Y"THENRETURN
2720 IFYN$<>"N"THEN2710
2730 END
2740 REM*****
2750 REM*MERGE RECORDS USING 4 DISC
2760 REM*FILES (2 IN 2 OUT). ON LAST
2770 REM*PASS WRITE TO OUTPUT FILE
2780 REM*****
2790 T1$="SORTFL,S,W":T2$="SORTFL,S,R":NF=INT((S0+1)/2)
2800 PRINT"START OF MERGE"
2810 F1$="A":F2$="B":F3$="C":F4$="D":LF=5:GOT02870
2820 WA$=F1$:F1$=F3$:F3$=WA$:WA$=F2$
2825 F2$=F4$:F4$=WA$:LF=5:PRINT"*";
2830 REM*****

```

```

2840 REM*OPEN 2 INPUT FILES & 1 OR 2
2850 REM*OUTPUT FILES
2860 REM*****
2870 OPEN3,8,3,F1$+T2$:IFST>0THENGOSUB2630
2880 OPEN4,8,4,F2$+T2$:IFST>0THENGOSUB2630
2890 IFNP=1THENF3$=OFFILE$:T1$=",SEQ,WRITE"
2900 OPEN5,8,5,F3$+T1$:IFST>0THENGOSUB2630
2910 IFNP=1THEN2930
2920 OPEN6,8,6,F4$+T1$:IFST>0THENGOSUB2630
2930 REM*****
2940 REM*READ FILES 3&4 &FORM FILES
2950 REM*5&6 (MERGED)
2960 REM*****
2970 IFI3#=Z4$THEN3000
2980 INPUT#3,I3$:IFST>0THENGOSUB2630
2990 K3#=MID$(I3$,SKSP,KLN)
3000 IFI4#=Z4$THENGOTO3030
3010 INPUT#4,I4$:IFST>0THENGOSUB2630
3020 K4#=MID$(I4$,SKSP,KLN)
3030 IFI3#=I4$ANDI3#=Z4$THENGOTO3330
3040 IFI3#>I4$THEN3230
3050 REM*****
3060 REM*AS I3#<=I4$ THEN W/O I3$ TO
3070 REM*LAST OUTPUT FILE,IF I3#<LAST
3080 REM*OUTPUT RECORD THEN CHANGE
3090 REM*OUTPUT FILES (CHANGE "LF"
3100 REM*****
3110 IFLF=6THEN3210
3120 IFK3#<K5$THEN3160
3130 PRINT#5,I3$:IFST>0THENGOSUB2630
3140 K5#=K3$:INPUT#3,I3$:IFST>0THENGOSUB2630
3150 K3#=MID$(I3$,SKSP,KLN):GOTO3030
3160 REM** START NEW STRING ON FILE 6**
3170 LF=6
3180 PRINT#6,I3$:IFST>0THENGOSUB2630
3190 K6#=K3$:INPUT#3,I3$:IFST>0THENGOSUB2630
3200 K3#=MID$(I3$,SKSP,KLN):GOTO3030
3210 IFK3#<K6$THENLF=5:GOTO3130
3220 GOTO3180
3230 REM*** WRITE I4$ TO OUTPUT **
3240 IFLF=6THEN3310
3250 IFK4#<K5$THEN3280
3260 PRINT#5,I4$:IFST>0THENGOSUB2630
3270 K5#=K4$:GOTO3000
3280 REM*** SWITCH OUTPUT FILES ***
3290 LF=6:PRINT#6,I4$:IFST>0THENGOSUB2630
3300 K6#=K4$:GOTO3000
3310 IFK4#<K6$THENLF=5:GOTO3260
3320 GOTO3290
3330 REM*****
3340 REM*END OF BOTH FILES. SCRATCH
3350 REM*INPUT FILES & W/O EOF
3360 REM*RECORDS ON OUT PUT FILES
3370 REM*****
3380 CLOSE3:IFST>0THENGOSUB2630
3390 CLOSE4:IFST>0THENGOSUB2630
3400 PRINT#10,"S:"+F1$+"SORTFL,"+F2$+"SORTFL":IFST>0THENGOSUB2630
3410 PRINT#5,Z4$:IFST>0THENGOSUB2630
3420 CLOSE5:IFST>0THENGOSUB2630
3430 IFNP=1THENRETURN

```

3440 PRINT#6,Z4\$:IFST>0THENGOSUB2630
3450 CLOSE6:IFST>0THENGOSUB2630
3460 NP=NP-1
3470 GOTO2820
READY.

```

1000 REM *****
1001 REM *PROGRAM TO MERGE A SORTED SEQUENTIAL
1002 REM *TRANSACTION FILE WITH A MASTER
1003 REM *DATA BASE SEQUENTIAL FILE. ALL DATA
1004 REM *FILES ARE ON DISK DRIVE ONE. THE
1005 REM *PROGRAM PRINTS CONTROL TOTALS OF
1006 REM *RECORD NUMBERS IN BOTH FILES.
1009 REM *****
1020 REM
1030 REM
1040 REM
1100 REM *** OPEN INPUT FILE ***
1102 REM
1104 OPEN2,8,2,"1:SORTED,SEQ,READ"
1106 IFST>0THEN1392
1108 REM
1110 REM *** OPEN OLD MASTER FILE ***
1112 REM
1114 OPEN3,8,3,"1:MASTER,SEQ,READ"
1116 IFST>0THEN1392
1118 REM
1120 REM *** OPEN NEW MASTER FILE **
1122 REM
1124 OPEN4,8,4,"@1:DUMMY,SEQ,WRITE"
1126 IFST>0THEN1392
1128 REM
1130 REM *** INITIALISE CONTROL TOTS ***
1132 REM
1134 BMB=0
1136 TMB=0
1138 CMB=0
1140 PRINT"MASTER FILE UPDATE PROGRAM"
1142 REM
1144 REM *** READ TRANSACTION***
1146 REM
1148 GOSUB1310
1150 REM
1152 REM *** READ OLD MASTER***
1154 REM
1156 GOSUB1346
1158 REM
1160 REM *** END OF UPDATE REACHED ? ***
1162 REM
1164 IFLEFT$(BREC$,3)="ZZZ"ANDLEFT$(TREC$,3)="ZZZ"THEN1206
1166 REM
1168 REM *** COMPARE OLD <> TRANS ***
1170 REM
1172 IFLEFT$(BREC$,16)<LEFT$(TREC$,16)THEN1190
1174 REM
1176 REM *** TRANS < OLD - INSERT ***
1178 REM
1180 PRINT#4,TREC$;CHR$(13);
1182 IFST>0THEN1392
1184 CMB=CMB+1
1186 GOSUB1310
1188 GOTO1164
1190 REM

```

```

1192 REM *** OLD < TRANS - CARRY FWD ***
1194 REM
1196 PRINT#4,BREC#;CHR$(13);
1198 IFST>0THEN1392
1200 CMB=CMB+1
1202 GOSUB1346
1204 GOTO1164
1206 REM
1208 REM *** END OF UPDATE ***
1210 REM
1212 PRINT#4,"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";CHR$(13);
1214 CLOSE4
1216 IFST>0THEN1392
1218 REM
1220 REM *** DISPLAY CONTROLS ***
1222 REM
1224 PRINT"MASTER FILE UPDATE"
1226 PRINT"OLD MASTER FILE RECORDS ";BMB
1228 PRINT"TRANSACTION RECORDS ";TMB
1230 PRINT"NEW MASTER FILE RECORDS ";CMB
1232 DISC=BMB+TMB-CMB
1234 IFDISC>0THENPRINT"WARNING RECORD DISCREPENCY";DISC
1236 REM
1238 REM *** CHECK UPDATE O.K. ***
1240 REM
1242 PRINT"IF UPDATE IS O.K. "
1244 PRINT"ENTER C TO CONTINUE ";
1246 GETA#;IFA#=""THEN1246
1248 IFA#="C"THEN1274
1250 INPUT"DO YOU WISH TO CANCEL THE UPDATE *";
1252 IFA#<>"Y"THEN1242
1254 REM
1256 REM *** SCRATCH OUTPUT MASTER ***
1258 REM
1260 OPEN1,8,15
1262 IFST>0THEN1392
1264 PRINT#1,"S1,DUMMY"
1266 IFST>0THEN1392
1268 CLOSE1
1270 IFST>0THEN1392
1272 GOTO1308
1274 REM
1276 REM *** SCRATCH INPUT FILES ***
1278 REM *** RENAME OUTPUT MASTER ***
1280 REM
1282 OPEN1,8,15
1284 IFST>0THEN1392
1286 PRINT#1,"S1:UNSORTED"
1288 PRINT#1,"S1:SORTED"
1290 PRINT#1,"S1:MASTER"
1292 PRINT#1,"R1:MASTER=1:DUMMY"
1294 IFST>0THEN1392
1296 CLOSE1
1298 IFST>0THEN1392
1300 CLOSE5
1302 REM
1304 REM *** END OF RUN ***
1306 REM
1308 POKE42,010:POKE43,36:CLR:LOAD"0:MENU",8
1310 REM

```

```

1312 REM *** READ TRANSACTION FILE ***
1314 REM
1316 IF LEFT$(TREC$,3)="ZZZ" THEN RETURN
1318 INPUT#2,TREC$
1320 IF LEFT$(TREC$,3)="ZZZ" THEN 1332
1322 IF ST>0 THEN 1392
1324 TMB=TMB+1
1328 PRINT"*****TRANSACTION RECORDS ";TMB
1330 RETURN
1332 REM
1334 REM *** END OF FILE ENCOUNTERED ***
1336 REM
1338 CLOSE2
1340 IF ST>0 THEN 1392
1342 PRINT"*****END OF TRANSACTIONS"
1344 RETURN
1346 REM
1348 REM *** READ OLD MASTER FILE ***
1350 REM
1352 IF LEFT$(BREC$,3)="ZZZ" THEN RETURN
1354 INPUT#3,BREC$
1356 IF LEFT$(BREC$,3)="ZZZ" THEN 1366
1358 BMB=BMB+1
1362 PRINT"*****OLD MASTER FILE RECORDS ";BMB
1364 RETURN
1366 REM
1368 REM *** END OF FILE ENCOUNTERED ***
1370 REM
1372 CLOSE3
1374 IF ST>0 THEN 1392
1376 PRINT"*****END OF OLD MASTER FILE"
1378 RETURN
1380 REM
1382 REM *** FATAL ERROR ***
1384 REM
1386 PRINT"*****FATAL ERROR ";MESS$
1388 GETA$:IFA$="" THEN 1388
1390 STOP
1392 REM
1394 REM *** DISC ERROR ***
1396 REM
1398 CLOSE1
1400 OPEN1,8,15
1402 INPUT#1,A$,B$,C$,D$
1404 IFA$="00" THEN RETURN
1406 PRINT"*****DISC ERROR "
1408 PRINTA$;" ";B$;" ";C$;D$
1410 GETA$:IFA$="" THEN 1410
1412 STOP
READY.

```

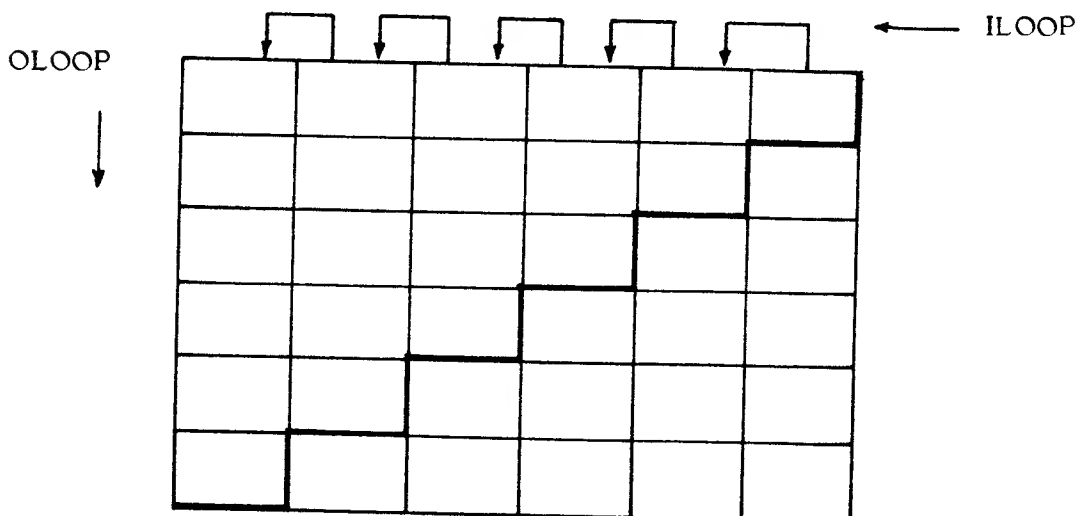
FAST MACHINE CODE SORT

This is a package comprising a 6502 machine code routine to perform a bubblesort on any specified one dimensional string array. The Basic subroutines allow this machine code program to be demonstrated on a specific array, these routines can be easily adapted to meet the users own requirements. The Basic subroutines provided include routines to input an array, sort it, add and delete strings, and locate strings using a binary chop search strategy.

The machine code sort searches the list of arrays until an array is found whose name matches a name previously stored in locations CA and CB. There are various failure exits: array not found (1), more than one dimension (2), and too few elements to sort (3). Once the named array is found the bubble sort commences. The pointers to each adjacent pair of strings are extracted and the strings compared. If they are in lexically incorrect order then the order of the pointers is reversed.

0,1 - 1,2 - 2,3 ----- N-1,N

This is performed up to N times in the inner loop ILOOP. every time a swap is made FLAG is set. If at the end of the inner loop FLAG is still clear, then the list is fully sorted and the routine terminates. FLAG is cleared at the beginning of each of the N iterations around the outer loop OLOOP. This outer loop is repeated until the list has been ordered by ILOOP N times, where there are N elements in the array, or until FLAG remains clear. The effect of the inner loop is to cause the lexically largest item in the array to move to the top of the array. Because of this the number of comparisons in the inner loop is reduced by one on each iteration of the outer loop.



When each string is being tested with its neighbour the corresponding character of each is compared, the first character of string one, with the first of string two, the second, third and so on. The first non-equal pair defines the lexical order. If they are equal, but of unequal length, such as, "there" and "therefore", the longer one is taken as lexically greater. Any null string (zero length, "") is taken as lexically infinite. The result of the sort is to leave the lexically smallest in element zero of the array, the non-null strings in ascending order; all the remaining elements are the null strings.

A more detailed description of the machine code program:

1st character of array name in CA, top bit cleared.
2nd character of array name in CB, top bit set. If array name is only 1 character long then CB=128.
The start of the array list is stored in 127,126, saved in BASE

LOCATE: If the first character of the array name is "\$" the array list is searched. ERROR=1 return (fail)

MOREA: If CA and CB both equal the array name, the array is found, goto AFOUND.

NEXTA: TEMP is loaded with the offset to next array descriptor block, TEMP is added to the current value of BASE, so that it points to the next array block. Goto LOCATE for next array.

AFOUND: Check array dimension equals one, if not ERROR = 2 and return (fail).

DOSORT: NOOFE is loaded with the number of elements in the array. If there are less than two elements then the array is unsortable, ERROR=3, return (fail).

INIT: NOOFE is transfered to NOOFC.

The outer loop:

OLOOP: Clear FLAG and COUNT. Decrement NOOFC, search to N-1 elements. If NOOFC is zero the array is sorted - return (success).

SPAIR: PAIR is set to point to the first byte of the block of six that contains the length and string pointers of the two strings to be compared.

The inner loop:

ILOOP: LEN1 set to length of first string. STR1 points to character in first string. LEN2 set to length of second string. STR2 points to character in second string. If LEN2 is zero then goto NOSWAP, because second string is greater or equal to first. If LEN1 is zero then goto SWAP, 1st string is null, second is not, so swap them.

COM: Compare each character of the two strings in turn, using the Y register. If char(str1) < char(str2) then goto SWAP.

NEXTCH: Increment Y register for next pair of characters. If it has overflowed then goto NOSWAP.

CKL2: If Y>LEN1 or Y>LEN2 then the longer is lexically greater, test by goto LENEQ. Otherwise compare next two characters back at COM.

LENEQ: If LEN1<=LEN2 then goto NOSWAP.

SWAP: Exchange length and string pointers in descriptor block: Length of string one set to LEN2. Pointer to string one set to STR2. Length of string two set to LEN1. Pointer to string two set to STR1. The exchange flag, FLAG, is set.

NOSWAP: Add one to COUNT, the number of pairs compared so far.

O1: If COUNT is not equal to NOOFC the inner loop is unfinished, goto PLUS3. If they are equal the inner loop is finished. If FLAG is still zero then no swaps were made and the sort is finished, return (success). Otherwise goto OLOOP to continue sorting.

PLUS3: Add three to PAIR to compare next pair of strings, goto ILOOP.

The Basic subroutines:

The Basic subroutines associated with the machine code sort program are intended to demonstrate its use. Throughout the Basic program all the variables used begin with "Z", any subroutine based around these routines should take this into account (Z1,ZB,ZM,ZN,ZP,ZT,ZA\$,ZI\$,ZS\$(),ZT\$ and ZY\$). Only a few of these are global, retaining useful values between subroutine calls, see below. All the examples assume the array to be manipulated is ZS\$(); besides this there are only three important variables:

Z1 - The base address of the machine code routines, all references to the machine code are made relative to this location. (480).

ZN - The number of elements (0 to ZN) in the default array ZS\$. (490,510).

ZS\$()- The default array. To change this all references to ZS\$() must be altered, see above, and statements 530-540 must be changed, or subroutine 400 evoked.

The Program.

Statement 1 lowers the top of memory to accomodate a machine code program at the top of RAM.

Statement 2 is the start address of the machine code program in decimal.

Statements 3-71 contain the machine code program in hexadecimal.

Statements 200-320 are the machine code loader.

Statement 480 sets Z1, alter if the machine code is relocated.

Statement 510 dimensions the various arrays that may be used.

Statements 530-540 default the array name "ZS" in locations CA and CB.

Statements 560-640, each of the demonstration subroutines may be called from a menu option pad, which is printed by subroutine 1000. The user types a digit which directs the system to one of the routines via a computed goto subroutine.

The option 'menu':

option

- 1 - Print the 'help' catalogue (1000)
- 2 - Input strings into the default array (2000)
- 3 - Print strings in the default array (3000)
- 4 - Change the sorted array name (4000)
- 5 - Sort the named array, default to ZS\$, (4500)
- 6 - Add a new string to the default array (5000)
- 7 - Delete a string from the default array (5500)
- 8 - Search for named string in default array (6000)

After performing each subroutine call, the user is prompted for his next choice. If in doubt "1" should be typed.

The subroutines.

1000 "MS3"

The 'help' option 'menu', clears the screen and prints the option list as previously described.

2000 "MS4"

Input strings into the default array.

a) All the strings in the default array are set to zero length (2060-2100).

b) A loop is entered (2140-2230), the user enters one string at a time (2190), prompted by the array number being filled (2180). At any time the user may leave the routine by typing a single "*", followed by the return key (2210). If at any time the available free memory of the machine falls below 256 bytes, the user is advised before the string prompt, but no further action is taken to protect the program against running out of memory (2160-2170).
 c) When the array is full the routine returns (2240). If the input was terminated prematurely the last input element is tidied (2250) and this routine returns (2270).

3000 "MS5"

Print the string in the default array on the PET screen.

a) A loop (3020-3060) is entered to print each of the array strings in turn (3044). If the array entry was empty a blank line is not printed (3030). ZM, previously set to zero, is a tally of non-null strings that have been printed (3050).
 b) The value of ZM is printed (3070), and the routine returns (3080).

4000 "MS6"

A subroutine to change the name of the array searched for and sorted by the machine code routine, pokes into locations CA and CB.

a) The name of the new array is requested (4010).
 b) Its last character must be "\$" for string array (4020), if it is not, a message is generated (4040) and a return effected (4050).
 c) The first character of the array name is extracted (4080) and checked as being alphabetic (4090), if it is not, a message is printed (4100) and the code requests the user to retype the string at a) (4110). If it is all right, the numeric value of the letter is poked into CA (4120). If there is no second character in the array name (4130), the routine returns, CB having previously been cleared (4060).
 d) The second character is extracted (4150), checked (4160-4180) as before, and if alphabetic poked into CB (4190) and the routine left (4200).

4500 "MS7"

A subroutine to call the machine code bubble sort routine.

a) The name of the array to be sorted is printed by peeking into CA and CB (4520).
 b) The machine code sort routine is called at Z1+14 (4530).
 c) The ERROR is read by peeking Z1+2 (4540).
 d) An appropriate message is generated if the sort was successful (4550), or failed (4560-4590), before the routine returns (4600).

5000 "MS8"

A subroutine to add a string into the default array.
a) The most probable place for an empty string (particularly after sorting) is the last position in the array. This is checked (5020). If it is not the user is invited to sort the array (5040), but not inside the routine, return at (5050). If this routine fails here after sorting, then the array is full.

b) A check is made on free space as in routine 2000 (5070), and a warning made if it is low (5080).

c) The user enters the new string (5100). Which is placed in the top array element.

d) The sort routine is called automatically (5140), before returning (5160).

5500 "MS9"

A subroutine to delete a string from the default array.

a) The user is asked to give the element number of the string to be erased (5520), if this is out of range (5530) a message is printed (5540) and the user may try again (5550).

b) If the string is already empty (5570), a message is printed (5590) and a return made (5600).

c) The selected string is printed (5620), to remind the user what is about to be erased. If the string is to be removed, the user types "Y" in response to (5640).

d) The sort routine is called automatically to tidy up the array (5690) before returning (5700-5710).

6000 "MS10"

A subroutine to search a sorted array for a target string input by the user; prints element number(s) and strings matched within the array, if any are found; uses the binary chop search strategy.

a) Sort the array - just to be sure (6020).

b) User inputs target string (6030).

c) User selects partial match ("Y") or not ("N") (6040). The partial match only characters in strings stored in the array to the length of the target string (ZT\$). If the target string were "THERE", it would find all strings beginning with "THERE", for instance "THERE", "THEREBY", "THEREFORE" etc. If partial match is not selected the match is for equality only, i.e. "THERE" only. In neither case would "THE" be matched.

d) ZT is set to the bottom of the array - 0 (6070), and ZB to the top (6080).

e) ZM is set to be half way between them, bisecting the array (6090).

f) ZI\$ is the ZMth string (6110).

g) If ZI\$ matches the target string, the search succeeds, goto step (j) to check forward and backward for other strings that match the target.

h) If ZM equals ZT then the array has been searched but no

match has been successful (6170). The routine returns printing a failure message (6460-6480).

- i) If the target is greater than the string at the bisect point, the bottom of the search space is set to ZM (6190), otherwise the top is set to ZM (6200). The search space for the routine is successively halved and may continue from step (e) (6210).
- j) Once a string is found, there may be others adjacent to it that will also match. Statements (6220-6300) step back through the array from ZM until the match fails (6280). ZB contains the lowest location that matched.
- k) Similarly statements (6310-6390) check forwards. ZT contains the highest location that matched.
- l) The element number and the string contained in that element of the array which matched with the target are printed in turn (6400-6440) before returning (6450). Subroutine 6600 truncates the length of the string in the array, if a partial match is wanted, to the length of the target string. It must be called before any match is attempted (6130,6270 and 6360)

LINE#	LOC	CODE	LINE
0001	0000		*****
0002	0000		PROGRAM TO PERFORM LEXICAL SORT
0003	0000		ON STRING ARRAY IN PET.
0004	0000		CA - CONTAINS FIRST LETTER
0005	0000		CB - SECOND LETTER OF STRING NAME
0006	0000		*****
0007	0000		
0008	0000		SYSTEM CONSTANTS
0009	0000		
0010	0000		ASTART =44
0011	0000		
0012	0000		ZERO PAGE LOCATIONS REQUIRED
0013	0000		
0014	0000		*=84
0015	0054		BASE *=+2
0016	0056		PAIR *=+2
0017	0058		STR1 *=+2
0018	005A		STR2 *=+2
0019	005C		
0020	005C		PROGRAM AND DATA AT TOP OF MEMORY
0021	005C		
0022	005C		*=32300
0023	7E2C		
0024	7E2C		ORDINARY VARIABLES AND STORAGE
0025	7E2C		
0026	7E2C		CA *=+1
0027	7E2D		CB *=+1
0028	7E2E		ERROR *=+1
0029	7E2F		NOOFE *=+2
0030	7E31		NOOFC *=+2
0031	7E33		LEN1 *=+1
0032	7E34		LEN2 *=+1
0033	7E35		COUNT *=+2
0034	7E37		TEMP *=+2
0035	7E39		FLAG *=+1
0036	7E3A		
0037	7E3A		THE CODE
0038	7E3A		CLEAR TOP BIT OF CA
0039	7E3A		
0040	7E3A	48	PHA
0041	7E3B	98	TYA
0042	7E3C	48	PHA
0043	7E3D	8A	TXA
0044	7E3E	48	PHA
0045	7E3F	AD 2C 7E	LDA CA
0046	7E42	29 7F	AND #20111111
0047	7E44	8D 2C 7E	STA CA
0048	7E47		
0049	7E47		SET TOP BIT OF CB
0050	7E47		
0051	7E47	AD 2D 7E	LDA CB
0052	7E4A	09 80	ORA #10000000
0053	7E4C	8D 2D 7E	STA CB
0054	7E4F		
0055	7E4F		CLEAR ERROR FLAG

LINE#	LOC	CODE	LINE
0056	7E4F		
0057	7E4F	A9 00	LDA #0
0058	7E51	8D 2E 7E	STA ERROR
0059	7E54		
0060	7E54		;SEARCH DOWN ARRAY NAME LIST
0061	7E54		;LOAD BASE WITH START OF ARRAYS
0062	7E54		
0063	7E54	A5 2C	LDA ASTART
0064	7E56	85 54	STA BASE
0065	7E58	A5 2D	LDA ASTART+1
0066	7E5A	85 55	STA BASE+1
0067	7E5C		
0068	7E5C		;LOCATE ARRAY NAMED IN CA & CB
0069	7E5C		
0070	7E5C	A0 00	LOCATE LDY #0
0071	7E5E	B1 54	LDA (BASE),Y
0072	7E60	C9 24	CMP #36 ;END OF ARRAY?
0073	7E62	D0 08	BNE MOREA ;NO
0074	7E64		
0075	7E64		;ERROR 1 - ARRAY NOT FOUND
0076	7E64		
0077	7E64	A9 01	LDA #1
0078	7E66	8D 2E 7E	STA ERROR
0079	7E69	4C AA 7F	JMP RTN
0080	7E6C		
0081	7E6C		;CHECK THIS ARRAY NAME
0082	7E6C		;AGAINST CA & CB
0083	7E6C		
0084	7E6C	CD 2C 7E	MOREA CMP CA ;1ST CHAR
0085	7E6F	D0 08	BNE NEXTA
0086	7E71	C8	INY ;NEXT CHAR
0087	7E72	B1 54	LDA (BASE),Y
0088	7E74	CD 2D 7E	CMP CB
0089	7E77	F0 1E	BEQ AFOUND ;THIS ARRAY
0090	7E79		
0091	7E79		;CHECK NEXT ARRAY IN TABLE
0092	7E79		;STORE RELATIVE OFFSET IN TEMP
0093	7E79		
0094	7E79	A0 02	NEXTA LDY #2
0095	7E7B	B1 54	LDA (BASE),Y
0096	7E7D	8D 37 7E	STA TEMP
0097	7E80	C8	INY
0098	7E81	B1 54	LDA (BASE),Y
0099	7E83	8D 38 7E	STA TEMP+1
0100	7E86		
0101	7E86		;ADD RELATIVE OFFSET TO BASE
0102	7E86		
0103	7E86	18	CLC
0104	7E87	A5 54	LDA BASE
0105	7E89	6D 37 7E	ADC TEMP
0106	7E8C	85 54	STA BASE
0107	7E8E	A5 55	LDA BASE+1
0108	7E90	6D 38 7E	ADC TEMP+1
0109	7E93	85 55	STA BASE+1
0110	7E95		

LINE#	LOC	CODE	LINE
0111	7E95		;NEXT ARRAY
0112	7E95		;
0113	7E95	90 C5	BCC LOCATE
0114	7E97		;
0115	7E97		;NAMED ARRAY FOUND
0116	7E97		;CHECK ONLY ONE DIMENSION
0117	7E97		;
0118	7E97	A0 04	AFOUND LDY #4
0119	7E99	B1 54	LDA (BASE),Y
0120	7E9B	C9 01	CMP #1
0121	7E9D	F0 08	BEQ DOSORT
0122	7E9F		;
0123	7E9F		;ERROR 2 - INCORRECT DIMENSION
0124	7E9F		;
0125	7E9F	A9 02	LDA #2
0126	7EA1	8D 2E 7E	STA ERROR
0127	7EA4	4C AA 7F	JMP RTN
0128	7EA7		;
0129	7EA7		;DO BUBBLE SORT ON ARRAY
0130	7EA7		;SAVE NUMBER OF ELEMENTS IN NOOFE
0131	7EA7		;
0132	7EA7	A0 05	DOSORT LDY #5
0133	7EA9	B1 54	LDA (BASE),Y
0134	7EAB	8D 30 7E	STA NOOFE+1
0135	7EAE	C8	INY
0136	7EAF	B1 54	LDA (BASE),Y
0137	7EB1	8D 2F 7E	STA NOOFE
0138	7EB4		;
0139	7EB4		;CHECK THERE ARE 2 OR MORE
0140	7EB4		;ELEMENTS IN ORDER
0141	7EB4		;
0142	7EB4	AE 30 7E	LDX NOOFE+1
0143	7EB7	D0 0F	BNE INIT ;MORE THAN 255
0144	7EB9	AD 2F 7E	LDA NOOFE
0145	7EBC	C9 02	CMP #2
0146	7EBE	B0 08	BCS INIT
0147	7EC0		;
0148	7EC0		;ERROR 3 - TOO FEW ELEMENTS
0149	7EC0		;
0150	7EC0	A9 03	LDA #3
0151	7EC2	8D 2E 7E	STA ERROR
0152	7EC5	4C AA 7F	JMP RTN
0153	7EC8		;
0154	7EC8		;SET UP SORT PARAMETERS
0155	7EC8		;
0156	7EC8	AD 2F 7E	INIT LDA NOOFE
0157	7ECB	8D 31 7E	STA NOOFC
0158	7ECE	AD 30 7E	LDA NOOFE+1
0159	7ED1	8D 32 7E	STA NOOFC+1
0160	7ED4		;
0161	7ED4		;OUTER SORT LOOP
0162	7ED4		;I=1 TO NOOFE-1
0163	7ED4		;CLEAR TO SWAP FLAGS
0164	7ED4		;
0165	7ED4	A9 00	OLOOP LDA #0

LINE#	LOC	CODE	LINE
0166	7ED6	8D 39 7E	STA FLAG
0167	7ED9		
0168	7ED9		CLEAR NO OF TESTS COUNTER
0169	7ED9		
0170	7ED9	8D 35 7E	STA COUNT
0171	7EDC	8D 36 7E	STA COUNT+1
0172	7EDF		
0173	7EDF		SUBTRACT ONE FROM NOOFC
0174	7EDF		NO OF COMPARISONS (1 LESS EACH TIME)
0175	7EDF		
0176	7EDF	38	SEC
0177	7EE0	AD 31 7E	LDA NOOFC
0178	7EE3	E9 01	SBC #1
0179	7EE5	8D 31 7E	STA NOOFC
0180	7EE8	AD 32 7E	LDA NOOFC+1
0181	7EEB	E9 01	SBC #1
0182	7EED	8D 32 7E	STA NOOFC+1
0183	7EF0		
0184	7EF0		ALL DONE WHEN NOOFC = 0
0185	7EF0		
0186	7EF0	AE 32 7E	LDX NOOFC+1
0187	7EF3	D0 0A	BNE SPAIR
0188	7EF5	AD 31 7E	LDA NOOFC
0189	7EF8	C9 00	CMP #0
0190	7EFA	D0 03	BNE SPAIR
0191	7EFC		
0192	7EFC		SORT FINISHED
0193	7EFC		
0194	7EFC	4C AA 7F	JMP RTN
0195	7EFF		
0196	7EFF		START OF STRING POINTERS
0197	7EFF		SPAIR = BASE+7
0198	7EFF		
0199	7EFF	18	SPAIR CLC
0200	7F00	A5 54	LDA BASE
0201	7F02	69 07	ADC #7
0202	7F04	85 56	STA PAIR
0203	7F06	A5 55	LDA BASE+1
0204	7F08	69 00	ADC #0
0205	7F0A	85 57	STA PAIR+1
0206	7F0C		
0207	7F0C		INNER LOOP - SUCCESSIVE COMPARISONS
0208	7F0C		
0209	7F0C	A0 00	ILOOP LDY #0
0210	7F0E	B1 56	LDA (PAIR),Y
0211	7F10	8D 33 7E	STA LEN1
0212	7F13	C8	INY
0213	7F14	B1 56	LDA (PAIR),Y
0214	7F16	85 58	STA STR1
0215	7F18	C8	INY
0216	7F19	B1 56	LDA (PAIR),Y
0217	7F1B	85 59	STA STR1+1
0218	7F1D	C8	INY
0219	7F1E	B1 56	LDA (PAIR),Y
0220	7F20	8D 34 7E	STA LEN2

LINE#	LOC	CODE	LINE
0221	7F23	C8	INY
0222	7F24	B1 56	LDA (PAIR),Y
0223	7F26	85 5A	STA STR2
0224	7F28	C8	INY
0225	7F29	B1 56	LDA (PAIR),Y
0226	7F2B	85 5B	STA STR2+1
0227	7F2D		
0228	7F2D		;COMPARE TWO ADJACENT STRINGS
0229	7F2D		;POINTED AT BY STR1 AND STR2
0230	7F2D		;IF LEN (STR2)=0 THEN NOSWAP
0231	7F2D		
0232	7F2D	AE 34 7E	LDX LEN2
0233	7F30	F0 58	BEQ NOSWAP
0234	7F32		
0235	7F32		;OTHERWISE IF LENGTH (STR1)=0
0236	7F32		;THEN SWAP
0237	7F32		
0238	7F32	AE 33 7E	LDX LEN1
0239	7F35	F0 2D	BEQ SWAP
0240	7F37		
0241	7F37		;CHECK EACH CHAR IN TURN
0242	7F37		;FIRST NON EQUAL PAIR
0243	7F37		;DEFINES LEXICAL ORDER
0244	7F37		
0245	7F37	A0 00	LDY #0
0246	7F39	B1 58	COM LDA (STR1),Y
0247	7F3B	D1 5A	CMP (STR2),Y
0248	7F3D	F0 05	BEQ NEXTCH ;EQUAL - NEXT CHAR
0249	7F3F	90 49	BCC NOSWAP ;STR1<STR2
0250	7F41	4C 64 7F	JMP SWAP ;STR1>STR2
0251	7F44	C8	NEXTCH INY ;NEXT 2 CHARS
0252	7F45	F0 43	BEQ NOSWAP ;255 CHARS EQUAL
0253	7F47	CC 33 7E	CPY LEN1 ;END OF STRING?
0254	7F4A	90 04	BCC CKL2 ;NO
0255	7F4C	F0 09	BEQ LENEQ
0256	7F4E	B0 07	BCS LENEQ
0257	7F50	CC 34 7E	CKL2 CPY LEN2
0258	7F53	90 E4	BCC COM
0259	7F55	F0 00	BEQ LENEQ
0260	7F57		
0261	7F57		;CHARS EQUAL TO LEN1 OR LEN2
0262	7F57		;LONGER STRING IS LEXICAL GREATER
0263	7F57		
0264	7F57	AD 33 7E	LENEQ LDA LEN1
0265	7F5A	CD 34 7E	CMP LEN2
0266	7F5D	F0 2B	BEQ NOSWAP ;EXACTLY EQUAL
0267	7F5F	90 29	BCC NOSWAP ;STR1 SHORTER
0268	7F61	4C 64 7F	JMP SWAP
0269	7F64		
0270	7F64		;REVERSE ORDER OF STRINGS BY
0271	7F64		;SWAPPING POINTERS ROUND
0272	7F64		
0273	7F64	A0 00	SWAP LDY #0
0274	7F66	AD 34 7E	LDA LEN2
0275	7F69	91 56	STA (PAIR),Y

LINE#	LOC	CODE	LINE
0276	7F6B	C8	INY
0277	7F6C	A5 5A	LDA STR2
0278	7F6E	91 56	STA (PAIR),Y
0279	7F70	C8	INY
0280	7F71	A5 5B	LDA STR2+1
0281	7F73	91 56	STA (PAIR),Y
0282	7F75	C8	INY
0283	7F76	AD 33 7E	LDA LEN1
0284	7F79	91 56	STA (PAIR),Y
0285	7F7B	C8	INY
0286	7F7C	A5 58	LDA STR1
0287	7F7E	91 56	STA (PAIR),Y
0288	7F80	C8	INY
0289	7F81	A5 59	LDA STR1+1
0290	7F83	91 56	STA (PAIR),Y
0291	7F85		
0292	7F85		;SET SWAP FLAG
0293	7F85		
0294	7F85	A9 01	LDA #1
0295	7F87	8D 39 7E	STA FLAG
0296	7F8A		
0297	7F8A		;ADD ONE TO COUNT
0298	7F8A		
0299	7F8A	EE 35 7E	NOSWAP INC COUNT
0300	7F8D	D0 03	BNE 01
0301	7F8F	EE 36 7E	INC COUNT+1
0302	7F92		
0303	7F92		;CHECK IF COUNT=NOOFC
0304	7F92		
0305	7F92	AD 35 7E	01 LDA COUNT
0306	7F95	CD 31 7E	CMP NOOFC
0307	7F98	D0 16	BNE PLUS3
0308	7F9A	AD 36 7E	LDA COUNT+1
0309	7F9D	CD 32 7E	CMP NOOFC+1
0310	7FA0	D0 0E	BNE PLUS3
0311	7FA2		
0312	7FA2		;INNER LOOP FINISHED
0313	7FA2		;IF NO SWAPS THEN ALL SORTED
0314	7FA2		
0315	7FA2	AE 39 7E	LDX FLAG
0316	7FA5	F0 03	BEQ RTN
0317	7FA7	4C D4 7E	JMP OLOOP
0318	7FAA	68	RTN PLA
0319	7FAB	AA	TAX
0320	7FAC	68	PLA
0321	7FAD	A8	TAY
0322	7FAE	68	PLA
0323	7FAF	60	RTS
0324	7FB0		
0325	7FB0		;ADD 3 TO PAIR FOR NEXT TWO
0326	7FB0		;STRINGS IN INNER LOOP
0327	7FB0		
0328	7FB0	18	PLUS3 CLC
0329	7FB1	A5 56	LDA PAIR
0330	7FB3	69 03	ADC #3

LINE#	LOC	CODE	LINE
00331	7FB5	85 56	STA PAIR
00332	7FB7	A5 57	LDA PAIR+1
00333	7FB9	69 00	ADC #0
00334	7FBB	85 57	STA PAIR+1
00335	7FBD	4C 00 7F	JMP ILOOP
00336	7FC0		.END

ERRORS = 0000

SYMBOL TABLE

SYMBOL	VALUE						
AFOUND	7E97	ASTART	002C	BASE	0054	CA	7E2C
CB	7E2D	CKL2	7F50	COM	7F39	COUNT	7E35
DOSORT	7EA7	ERROR	7E2E	FLAG	7E39	ILOOP	7F0C
INIT	7EC8	LEN1	7E33	LEN2	7E34	LENEQ	7F57
LOCATE	7E5C	MOREA	7E6C	NEXTA	7E79	NEXTCH	7F44
NOOFC	7E31	NOOFE	7E2F	NOSWAP	7F8A	O1	7F92
OLOOP	7ED4	PAIR	0056	PLUS3	7FB0	RTN	7FAA
SPAIR	7EFF	STR1	0058	STR2	005A	SWAP	7F64
TEMP	7E37						

END OF ASSEMBLY

```

1 REM *****
2 REM *BASIC LOADER FOR MACHINE CODE
3 REM *SORT PROGRAM. LOADS INTO MEMORY
4 REM *FROM 32314 UP. TOP OF MEMORY
5 REM *IS PROTECTED (LINE 10).
9 REM *****
10 POKE52,00:POKE53,120
20 DATA32314
21 DATA48,98,48,8A,48
22 DATAAD,2C,7E,29,7F,8D,2C,7E
24 DATAAD,2D,7E,09,80,8D,2D,7E
26 DATAA9,00,8D,2E,7E
28 DATAA5,2C,85,54,A5,2D,85,55
30 DATAA0,00,B1,54,C9,24,D0,08
32 DATAA9,01,8D,2E,7E,4C,AA,7F
34 DATACD,2C,7E,D0,08,C8,B1,54,CD,2D,7E,F0,1E
36 DATAA0,02,B1,54,8D,37,7E,C8,B1,54,8D,38,7E
38 DATA18,A5,54,6D,37,7E,85,54,A5,55,6D,38,7E,85,55
40 DATA90,C5
42 DATAA0,04,B1,54,C9,01,F0,08
44 DATAA9,02,8D,2E,7E,4C,AA,7F
46 DATAA0,05,B1,54,8D,30,7E,C8,B1,54,8D,2F,7E
48 DATAAE,30,7E,D0,0F,AD,2F,7E,C9,02,B0,08
50 DATAA9,03,8D,2E,7E,4C,AA,7F
52 DATAAD,2F,7E,8D,31,7E,AD,30,7E,8D,32,7E
54 DATAA9,00,8D,39,7E
56 DATA8D,35,7E,8D,36,7E
58 DATA38,AD,31,7E,E9,01,8D,31,7E,AD,32,7E,E9,00,8D,32,7E
60 DATAAE,32,7E,D0,0A,AD,31,7E,C9,00,D0,03
62 DATA4C,AA,7F
64 DATA18,A5,54,69,07,85,56,A5,55,69,00,85,57
66 DATAA0,00,B1,56,8D,33,7E,C8,B1,56,85,58,C8,B1,56,85,59,C8
68 DATAB1,56,8D,34,7E,C8,B1,56,85,5A,C8,B1,56,85,5B
70 DATAAE,34,7E,F0,58
72 DATAAE,33,7E,F0,2D
74 DATAA0,00,B1,58,D1,5A,F0,05,90,49,4C,64,7F,C8
76 DATAF0,43,CC,33,7E,90,04,F0,09,B0,07,CC,34,7E,90,E4,F0,00
78 DATAAD,33,7E,CD,34,7E,F0,2B,90,29,4C,64,7F
80 DATAA0,00,AD,34,7E,91,56,C8,A5,5A,91,56,C8,A5,5B,91,56,C8
82 DATAAD,33,7E,91,56,C8,A5,58,91,56,C8,A5,59,91,56
84 DATAA9,01,8D,39,7E
86 DATAEE,35,7E,D0,03,EE,36,7E
88 DATAAD,35,7E,CD,31,7E,D0,16,AD,36,7E,CD,32,7E,D0,0E
90 DATAAE,39,7E,F0,03,4C,D4,7E,68,AA,68,A8,68,60
92 DATA18,A5,56,69,03,85,56,A5,57,69,00,85,57,4C,0C,7F
94 DATA*
200 READL
210 READA$
220 C=LEN(A$)
230 IFA$="*"THEN400
240 IFC<10RC>2THEN320
250 A=ASC(A$)-48
260 B=ASC(RIGHT$(A$,1))-48
270 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
280 IFN<00ORN>255THEN320
290 POKEL,N
300 L=L+1
310 GOTO210
320 PRINT"BYTE"L="[A$]" ???"
READY.

```

```

400 REM *****
410 REM *PROGRAM TO DEMONSTRATE SORT/SEARCH
420 REM *ADD/DELETE ON STRING ARRAYS
430 REM *DEMONSTRATION ARRAY IS Z$( )
440 REM *****
450 REM
460 REM
470 REM **BASE ADDRESS OF MACHINE CODE**
480 Z1=32300 414 = 733A
490 INPUT "HOW MANY ELEMENTS";ZN
500 REM **RESERVE ARRAY SPACE**
510 DIM Z$(1),ZP$(1),ZS$(ZN)
520 REM **DEFAULT ARRAY NAME**
530 POKEZ1,ASC("Z")
540 POKEZ1+1,ASC("S")
550 REM **PRINT HELP**
560 GOSUB1000
570 INPUT"OPTION";ZO
580 IFZO>4THEN610
590 ONZO GOSUB1000,2000,3000,4000
600 GOTO570
610 IFZO>8THEN640
620 ONZO-4 GOSUB4500,5000,5500,6000
630 GOTO570
640 PRINT"?????"
650 GOTO570
READY.

```

```

1000 REM *****
1010 REM *ROUTINE TO PRINT MENU
1020 REM *OF FUNCTIONS IN PACKAGE
1030 REM *ON SCREEN.
1040 REM *****
1050 PRINT"OPTION"
1060 PRINT"1-HELP"
1070 PRINT"2-INPUT"
1080 PRINT"3-PRINT"
1090 PRINT"4-NAME ARRAY"
1100 PRINT"5-SORT"
1110 PRINT"6-ADD STRING"
1120 PRINT"7-DELETE STRING"
1130 PRINT"8-SEARCH"
1140 RETURN
READY.

```

```

2000 REM *****
2010 REM *SUBROUTINE TO INPUT STRING INTO
2020 REM *BASIC ARRAY FROM KEYBOARD
2030 REM *RECODE FOR DIFFERENT DEVICES
2040 REM *DISK, TAPE, IEEE ETC
2050 REM *STRINGS INTO ZS$(0 TO ZN)
2060 REM *****
2070 REM
2080 REM
2090 PRINT:PRINT
2100 REM **CLEAR ALL STRINGS**
2110 FORZI=0TOZN
2120 ZS$(ZI)=""
2130 REM
2140 NEXTZI
2150 PRINT"INPUT ONE STRING AT A TIME"
2160 PRINT"TYPE '/' + RETURN TO TERMINATE"
2170 PRINT
2180 FORZI=0TOZN
2190 REM **CHECK FREE SPACE AVAILABLE**
2200 IF FRE(0)>256 THEN 2220
2210 PRINT"WARNING - ONLY ";FRE(0);"BYTES LEFT!"
2220 PRINT"STRING";ZI
2230 INPUTZS$(ZI)
2240 REM
2250 IFZS$(ZI)="/"THEN 2290
2260 REM
2270 NEXTZI
2280 RETURN
2290 ZS$(ZI)=""
2300 REM
2310 RETURN
READY.

```

```

3000 REM *****
3001 REM *SUBROUTINE TO PRINT ZS$ ON SCREEN
3009 REM *****
3010 ZM=0
3020 FORZI=0TOZN
3030 IFZS$(ZI)=""THEN 3080
3040 REM
3050 PRINTZS$(ZI)
3060 REM
3070 ZM=ZM+1
3080 NEXTZI
3090 PRINT"NUMBER OF ENTRIES IS ";ZM
3100 RETURN
READY.

```

```

4000 REM *****
4001 REM *SUBROUTINE TO NAME ARRAY TO
4002 REM *BE SORTED.
4009 REM *****
4010 INPUT "ARRAY NAME";ZT$
4020 REM **TEST TO CHECK STRING ARRAY**
4030 IF RIGHT$(ZT$,1)="$"THEN4060
4040 PRINT"STRING ARRAY ONLY"
4050 RETURN
4060 POKE Z1+1,0
4070 REM **1ST CHARACTER**
4080 ZA$=LEFT$(ZT$,1)
4090 IFZA$>="A" AND ZA$<="Z"THEN 4120
4100 PRINT"ALPHABETIC NAME ONLY - 1ST CHAR"
4110 GOTO4010
4120 POKE Z1,ASC(ZA$)
4130 IFLEN(ZT$)<=2 THEN RETURN
4140 REM **2ND CHARACTER, IF ANY**
4150 ZA$=MID$(ZT$,2,1)
4160 IF ZA$>="A" AND ZA$ <="Z" THEN 4190
4170 PRINT"ALPHABETIC NAME ONLY - 2ND CHAR"
4180 GOTO4010
4190 POKE Z1+1,ASC(ZA$)
4200 RETURN
READY.

```

```

4500 REM *****
4501 REM *SUBROUTINE TO SORT NAMED ARRAY
4502 REM *USING MACHINE CODE ROUTINE
4509 REM *****
4510 REM
4520 PRINT"SORTING ARRAY ";CHR$(PEEK(Z1));CHR$(PEEK(Z1+1));"$"
4530 SYS(Z1+14)
4540 ZI=PEEK(Z1+2)
4550 IFZI=0 THEN PRINT"SORTED O.K."
4560 IFZI=1 THEN PRINT"ARRAY NOT FOUND"
4570 IFZI=2 THEN PRINT"NOT ONE DIMENSION ARRAY"
4580 IFZI=3 THEN PRINT"TOO FEW ELEMENTS"
4590 IFZI>3 THEN PRINT"UNSPECIFIED SORT ERROR!"
4600 RETURN
READY.

```

```

5000 REM *****
5001 REM *ADD A STRING INTO ZS$
5002 REM *CHECK LOCATION FREE
5009 REM *****
5010 REM
5020 IF ZS$(ZN)="" THEN 5060
5030 REM
5040 PRINT "NO SPACE - TRY SORT"
5050 RETURN
5060 REM **CHECK SUFFICIENT SPACE FREE**
5070 IF FRE(0)>255 THEN 5100
5080 PRINT "ONLY ";FRE(0);"BYTES LEFT!"
5090 REM **GET STRING TO ADD**
5100 PRINT "INPUT STRING"
5110 INPUT ZS$(ZN)
5120 REM
5130 REM **SORT**
5140 SYS(ZI+14)
5150 PRINT "O.K."
5160 RETURN
READY.

```

```

5500 REM *****
5501 REM *DELETE STRING IN ZS$
5502 REM *REQUEST STRING NUMBER
5509 REM *****
5510 REM
5520 INPUT "STRING NUMBER";ZI
5530 IF ZI>=0 AND ZI<=ZN THEN 5570
5540 PRINT ZI;"OUT OF RANGE"
5550 GOTO 5520
5560 REM **DONT DELETE NULL STRINGS**
5570 IF ZS$(ZI)<>"" THEN 5620
5580 REM
5590 PRINT "NULL STRING ALREADY"
5600 RETURN
5620 PRINT "STRING";ZI;" IS ";ZS$(ZI);"/"
5630 REM
5640 INPUT "DELETE (Y/N)";ZY$
5650 IF LEFT$(ZY$,1)<>"Y" THEN RETURN
5660 ZS$(ZI)=""
5670 REM
5680 REM **AND SORT**
5690 SYS(ZI+14)
5700 PRINT "O.K."
5710 RETURN
READY.

```

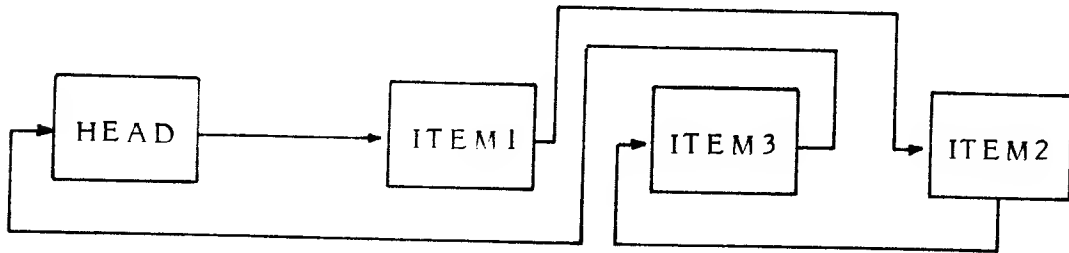
```

6000 REM *****
6001 REM *SUBROUTINE TO BINARY SEARCH
6002 REM *ARRAY Z$().
6003 REM *****
6010 REM **SORT TO BE SURE**
6020 SYS(Z1+14)
6030 INPUT "TARGET STRING"; ZT$
6040 INPUT "PARTIAL MATCH (Y/N)"; ZY$
6050 ZP=1
6060 IF LEFT$(ZY$,1)="Y" THEN ZP=0
6070 ZT=0
6080 ZB=ZN+1
6090 ZM=ZT+INT((ZB-ZT)/2)
6110 ZI$=ZS$(ZM)
6120 REM
6130 GOSUB 6600
6140 REM **THE COMPARISONS**
6150 IF ZI$=ZT$ THEN 6220
6160 REM **STRING NOT FOUND**
6170 IF ZT=ZM THEN 6460
6180 REM **CHANGE POINTERS**
6190 IF ZI$="" OR ZT$>ZI$ THEN ZT=ZM: GOTO 6090
6200 ZB=ZM
6210 GOTO 6090
6220 REM **FOUND - CHECK BACK FOR OTHERS**
6230 ZB=ZM
6240 IF ZB-1<0 THEN 6310
6250 ZI$=ZS$(ZB-1)
6260 REM
6270 GOSUB 6600
6280 IF ZI$<>ZT$ THEN 6310
6290 ZB=ZB-1
6300 GOTO 6240
6310 REM **CHECK FORWARD FOR OTHERS**
6320 ZT=ZM
6330 IF ZT+1>ZN THEN 6400
6340 ZI$=ZS$(ZT+1)
6350 REM
6360 GOSUB 6600
6370 IF ZI$<>ZT$ THEN 6400
6380 ZT=ZT+1
6390 GOTO 6330
6400 REM **PRINT ALL MATCHED STRINGS**
6410 FOR ZI=ZB TO ZT
6420 PRINT ZI; " "; ZS$(ZI)
6430 REM
6440 NEXT ZI
6450 RETURN
6460 REM **STRING NOT FOUND**
6470 PRINT "STRING NOT FOUND"
6480 RETURN
6600 REM **STRING MANIPULATION**
6610 IF ZP=0 AND (LEN(ZI$)>LEN(ZT$)) THEN ZI$=LEFT$(ZI$,LEN(ZT$))
6620 RETURN
READY.

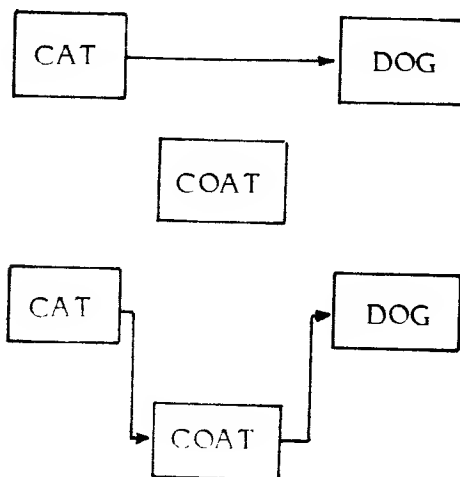
```

SORTING WITH A LINKED LIST

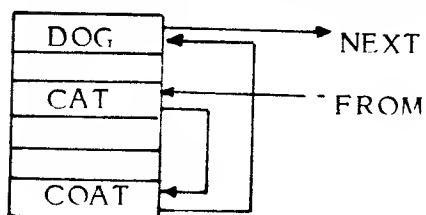
A linked list is a way of organising data in sorted order without the data having to be physically in sequence, the data is instead ordered logically. The data is organised using pointers associated with each data item, this allows data to be sorted without physically moving the data. This technique is commonly used with disk files and may be used internally as a sorting technique.



If data is held in a table like this, a user can output the data either in the sequence in which it is held or in its chained sequence. Each item input is compared with items in the chain, starting from the head of the chain, and is linked into its correct position.



In a table this might appear as:



The head of the list which points to the first data item in the chain is held in the first location in the table. The last item in the chain points back to the head of the chain. The program puts each data item in the next free place in the table and links it into its place in the chain.

LINKLIST

This is a set of four integrated subroutines designed to create a linked list data array and read/write to that array.

24008 - Initialise: this routine creates the head and tail pointers for a new list.

24026 - Input to list: this initialises the variables and calls the routines to get text and pointers and insert new item in array.

24048 - Get text and pointer: this routine gets the text in PL\$(PP) and extracts the pointer from the end of the text which it returns as variable PR. Variable PP is set to 1 to access the head of the file.

24076 - Insert text: this routine is only used by the routine at 24028, it takes the input string P\$ finds its correct position in the list using the subroutine at 24048 and adds the correct pointers on to the end of the string, using the back slash character as an end of string marker.

Parameters used:

P\$ - string variable to be input to linked list.
PL\$()- array in which linked list is stored.
PL - number of elements in array PL\$().
FL - pointer to next free element in array PL\$().
PI - error flag, if 1 then array is full.
PP - element in array to be accessed by routine 24048.
PR - pointer to next element in chain returned by routine 24048.

```

100 PL=100: REM **NUMBER OF ELEMENTS IN LIST**
110 FL=1: REM **NEXT FREE ELEMENT IN LIST**
120 DIMPL$(PL)
1000 REM *****
1001 REM *EXAMPLE OF WAY LINK LIST ROUTINES
1002 REM *CAN BE USED TO STORE AND RETREIVE
1003 REM *DATA.
1009 REM *****
1010 PRINT"DATALINK LIST DEMO PROGRAM FUNCTIONSDATA"
1020 PRINT"1-INPUT DATA TO LINK LIST"
1030 PRINT"2-PRINT DATA IN CHAINED ORDER"
1040 PRINT"3-PRINT DATA AS HELD"
1050 PRINT"4-END"
1060 GETA$: IFA$="" THEN 1060
1070 A=VAL(A$)
1080 ONAGOTO1100,1300,1500,1090
1090 END
1100 REM **INPUT DATA TO LIST**
1110 PRINT"DATAINPUT DATA TO LINK LIST"
1120 PRINT"TO END ENTER 'END'DATA"
1130 INPUTP$
1140 IFF$="END" THEN 1010
1150 GOSUB24000
1160 GOTO1130
1300 REM **PRINT DATA IN CHAINED ORDER**
1310 PRINT"DATADATA IN CHAINED SEQUENCEDATA"
1320 PP=1
1330 GOSUB24050
1340 PP=PR
1350 GOSUB24050
1360 PRINTA1$
1370 IFPR>1 THEN 1340
1380 GET A$: IFA$="" THEN 1380
1390 GOTO1010
1500 REM **PRINT DATA IN LIST SEQUENCE**
1510 PRINT"DATADATA IN LIST SEQUENCEDATA"
1520 PRINT"ELEMENT POINTER DATADATA"
1530 FORQ=1 TO FL-1
1540 PP=Q
1550 GOSUB24050
1560 PRINTQ,PR,A1$
1570 NEXTQ
1580 GET A$: IFA$="" THEN 1380
1590 GOTO1010
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
24000 REM*****
24002 REM*SUBROUTINES TO SET UP AND
24004 REM*ACCESS LINK LIST ARRAY.
24006 REM*****
24008 REM
24010 REM***INITIALISE LINK LIST***
24012 REM
24014 IFFL>1 THEN 24032

```

```

24016 PL$(1)=" \2"
24018 FH=2
24020 PL$(2)=P$+"\1"
24022 FL=3
24024 RETURN
24026 REM
24028 REM*****INPUT TO LIST*****
24030 REM
24032 PP=1:P1=0
24034 IFFLD>=PL+1THENP1=1:GOTO24046
24036 REM
24038 REM
24040 REM
24042 GOSUB24048
24044 GOSUB24076
24046 RETURN
24048 REM
24050 REM*****GET TEXT & POINTER***
24052 REM
24054 A2=LEN(PL$(PP))
24056 FORA3=1TOA2
24058 IFMID$(PL$(PP),A3,1)="\"THEN24062
24060 NEXTA3
24062 A1$=LEFT$(PL$(PP),A3-1)
24064 A2$=""
24066 FORA4=A3+1TOA2
24068 A2$=A2$+MID$(PL$(PP),A4,1)
24070 NEXTA4
24072 PR=VAL(A2$)
24074 RETURN
24076 REM
24078 REM*****INSERT TEXT*****
24080 REM
24082 A1=PP
24084 PP=PR
24086 GOSUB24048
24088 IFP$<A1$THEN24104
24090 IFP$=A1$ORPR=1THEN24098
24092 IFP$>A1$THEN24082
24094 PP=A1
24096 GOSUB24048
24098 PL$(PP)=A1$+"\ "+STR$(FL)
24100 PL$(FL)=P$+"\ "+A2$
24102 GOTO24112
24104 IFFH<>PPTHEN24094
24106 PL$(1)=" \ "+STR$(FL)
24108 PL$(FL)=P$+"\ "+STR$(FH)
24110 FH=FL
24112 FL=FL+1
24114 RETURN
READY.

```

SORTED OUTPUT

When the only requirement is to print out an array in sorted order, it is unnecessary to actually sort the array prior to printing. The sorting can be done as part of the array printing subroutine. The following subroutine is designed to print an array in sorted order. This subroutine is ideal if for example, all that is required is an alphabetic index of data stored either in core or on disk. Since indexing will account for many of the applications of this routine the option has been added to give a character heading to each alphabetic section.

PRINTSORT

This subroutine will output the contents of array P\$ in sorted order either on the screen or on a PET printer. The routine requests which output device is to be used, screen or printer or both. The user is then prompted as to whether section headings are required (a section heading would for example be A above all entries in the array beginning with the character A). The array may be printed in both sorted and unsorted order. It should be noted that when sorting data consisting of long data strings which include punctuation, unexpected results may appear, due to the character representation given in PET ASCII to punctuation and graphics characters. By changing the device numbers used in this subroutine it could be made to output data in sorted order to a disk drive; a simple way of ensuring that data on a sequential disk file is in alphanumeric order.

Parameters used:

P\$() - data storage array from which data is to be output in sorted order.

PT%()- tag array used by sort code to indicate which elements in array P\$() have been printed.

PN - number of elements in array P\$() which are to be sorted.

```

100 DIM P$(200),PT (200)
1000 REM *****
1001 REM *EXAMPLE OF SUBROUTINE CALL TO
1002 REM *PRINT OUT IN SORTED ORDER THE
1003 REM *CONTENTS OF ARRAY P$( ).
1009 REM *****
1010 FOR Q=1TO10
1020 INPUTP$(Q):REM **INPUT ARRAY P$( )**
1030 NEXTQ
1040 PN=10:REM **NUMBER OF RECORDS TO BE SORTED**
1050 GOSUB21000
1060 END
2000 REM
3000 REM
4000 REM
5000 REM
6000 REM
21000 REM *****
21002 REM *ROUTINE TO PRINT AN ARRAY IN
21004 REM *IN ALPHANUMERIC ORDER ON EITHER
21006 REM *THE PET SCREEN, PRINTER OR OTHER
21008 REM *IEEE I/O DEVICE.
21010 REM *THIS TECHNIQUE OFFERS ADVANTAGES
21012 REM *OVER SORTING THE LIST BEFORE
21014 REM *PRINTING, PARTICULARLY IN TERMS
21016 REM *OF TIME TAKEN.
21018 REM *NOTE: THIS ROUTINE SORTS ACCORDING
21020 REM *TO THE PET'S INTERNAL REPRESENTATION
21022 REM *OF THE ASCII CHARACTER SET,
21024 REM *PUNCTUATION MAY THEREFORE HAVE
21026 REM *SOME UNEXPECTED EFFECTS.
21028 REM *****
21030 PRINT"*****PRINTS STRINGS IN ALPHANUMERIC ORDER"
21032 PRINT"===== "
21034 REM **DETERMINE WHERE TO BE SENT**
21036 AS=0:AP=0
21038 INPUT"*****OUTPUT TO SCREEN (Y OR N)";Y$
21040 IF LEFT$(Y$,1)="Y" THEN AS=1
21042 INPUT"*****OUTPUT TO PRINTER (Y OR N)";Y$
21044 IF LEFT$(Y$,1)="Y" THEN AP=1
21046 REM **OPEN PRINTER CHANNEL**
21048 IFAP=0THEN21054
21050 CH=4: REM **PRINTER CHANNEL**
21052 OPENCH,CH
21054 REM **SECTION HEADINGS**
21056 INPUT"*****SECTION HEADINGS (Y OR N)";Y$
21058 AT=0
21060 IFLEFT$(Y$,1)<>"Y" THEN AT=1
21062 REM **PRINT UNSORTED LIST**
21064 INPUT"*****DO YOU WISH TO PRINT UNSORTED LIST ?";Y$
21066 IFLEFT$(Y$,1)<>"Y" THEN 21076
21068 FOR I=1TO PN
21070 IF AS=1 THEN PRINTP$(I)
21072 IFAP=1THEN PRINT#CH,P$(I)
21074 NEXTI
21076 REM **CLEAR 'PRINTED' FLAGS**
21078 FORI=1TO PN

```

```

21080 PT(I)=0
21082 NEXT I
21084 IF AS=1 THEN PRINT:PRINT:PRINT
21086 IF AP=1 THEN PRINT#CH:PRINT#CH:PRINT#CH
21088 REM **PRINT SMALLEST IN OUTER LOOP.**
21090 REM **TAG ONCE PRINTED, NEXT TIME**
21092 REM **ROUND PRINT SMALLEST REMAINING**
21094 AH$=CHR$(255)
21096 FOR I=1 TO PN
21098 AM$=CHR$(255)
21100 AM=1
21102 FOR J=1 TO PN
21104 IF PT(J)=1 THEN 21108
21106 IF P$(J)<AM$ THEN AM$=P$(J):AM=J
21108 NEXT J
21110 REM **TAGGED SMALLEST FOUND**
21112 PT(AM)=1
21114 REM **PRINT LETTER HEADING (IF REQD.)**
21116 AN$=LEFT$(AM$,1)
21118 IF((AH$=AN$)OR AT=1) THEN 21124
21120 IF AS=1 THEN PRINT:PRINTAN$:PRINT
21122 IF AP=1 THEN PRINT#CH:PRINT#CH,AN$:PRINT#CH
21124 REM **PRINT SMALLEST**
21126 IF AS=1 THEN PRINTAM$
21128 IF AP=1 THEN PRINT#CH,AM$
21130 AH$=AN$
21132 NEXT I
21134 PRINT:PRINT:PRINT:PRINT
21136 REM **ROUND AGAIN?**
21138 INPUT"DO YOU WISH TO PRINT LIST AGAIN";Y$
21140 IF LEFT$(Y$,1)="Y" THEN 21034
21142 PRINT"O.K."
21144 RETURN
READY.

```

SEQUENTIAL DATA FILES

Sequential access disk files are the simplest form of disk data storage. Sequential file access is however not easily incorporated into standard subroutines. The subroutines used in this section consist of simple function blocks. To read or write a file will require on average three subroutine calls, one of which is repeatedly used within a loop. The demonstration program from lines 100-1290 show how they are used to read or write a file. The sequence of operations used is as follows:

- a) At the beginning of the program the data disk should be initialised and the command channel opened using subroutine 24400. This must be done whether the file is to be written or read.
- b) To write a data file (file name F\$) first call subroutine 24000 to open a logical file for writing. This subroutine allows a new file to replace an old file, should an old file of the same name exist. Each record - P\$ - can then be written on to disk using subroutine 24200; repeat calling this subroutine until all records have been recorded. The logical file can then be closed using subroutine 24100.
- c) To read a sequential file (file name F\$) first open a logical file to read with routine 24050. Data can then be read by repeatedly calling subroutine 24300; each record is output as P\$. The end of the file is indicated by the variable P being set to 64; the number of records on the file is contained in the variable IS. The logical file is then closed using subroutine 24100.

The usual practice is to store the data for a sequential file in an array, though data can be input as individual records providing the file is kept open between writing each record. Data output can be either into an array or directly on to the screen or printer. These subroutines all use data in string format as the variable to be read or written to disk, this in my opinion is better than mixing data types on the file. This means that numerical data must be converted into string format before writing to disk, and vice versa when reading. Although the variable P\$ is used to transfer data between the disk file and the main program, successive records need not originate from the same variable. Variables A\$, B\$ and C\$ can be stored as successive records, each transferred as P\$ to the subroutine. When using successive sequences of different variables it is important always to retain that sequence by recording each record, even if it has a null entry, otherwise when reading the file one will get out of order. If the number of records in each sequence is variable, then an end of sequence marker record must be used.

Parameters used:

- CH - sequential access channel number, more than one value must be used if more than one file is accessed at a time. Typical value is 2 or 6
- D - drive number (either 0 or 1) of sequential data file.
- F\$ - file name of sequential data file to be accessed.
- P\$ - data as string variable to be stored on disk or read from disk.
- P - end of file flag normally = 0, if end of file then =64.
- IS - number of records read from a sequential file, should be set to 0 before accessing file. Can be used to determine relative position of target data on file.

```

100 CH=2:D=1:REM *CHANNEL AND DRIVE NUMBER**
110 SP$=""
120 CR$=CHR$(13): REM **CARRAGE RETURN*
1000 REM *****
1001 REM *DEMONSTRATION OF USE OF SEQ
1002 REM *FILE SUBROUTINES.
1009 REM *****
1010 GOSUB24400: REM *INITIALISE*
1020 PRINT"*****SEQUENTIAL DISK FILE DEMO FUNCTIONS*****"
1030 INPUT"INPUT FILE NAME";F$
1040 PRINT"001-CREATE DATA FILE"
1050 PRINT"002-READ DATA FILE"
1060 PRINT"003-END"
1070 GETA$: IFA$="" THEN 1070
1080 A=VAL(A$)
1090 ONAGOTO1110,1200,1100
1100 END
1110 REM *
1111 REM *CREATE DATA FILE.
1119 REM *
1120 GOSUB24000
1130 PRINT"*****INPUT DATA TO BE STORED ON DISK*****"
1140 INPUTP$
1150 IFF$="END" THEN GOSUB24100:GOTO1020
1160 GOSUB24200
1170 GOTO1140
1200 REM *
1201 REM *PRINT SEQUENTIAL FILE
1209 REM *
1210 PRINT"*****DISPLAY DATA ON FILE*****"
1220 GOSUB24050 :IS=0
1230 GOSUB24300
1240 PRINTIS,P$,P: REM *PRINT RECORD NUMBER, DATA, AND STATUS*
1250 IFF=64 THEN GOTO1270: REM *END OF FILE?*
1260 GOTO1230
1270 GETA$: IFA$="" THEN 1270
1275 GOSUB24100
1280 GOTO1020
2000 REM
3000 REM
4000 REM
5000 REM
24000 REM *****
24001 REM *OPEN LOGICAL FILE TO WRITE
24009 REM *****
24010 OPENCH(8,CH,"@"+STR$(D)+":"+F$+").S,W"
24020 GOSUB30300
24030 RETURN
24050 REM *****
24051 REM *OPEN LOGICAL FILE TO READ
24059 REM *****
24060 OPENCH(8,CH,STR$(D)+":"+F$+").S,R"
24070 GOSUB30300
24080 RETURN
24100 REM *****
24101 REM *CLOSE LOGICAL FILE
24109 REM *****

```

```

24110 CLOSECH
24120 RETURN
24200 REM *****
24201 REM *PRINT RECORD TO FILE
24209 REM *****
24210 PRINT#CH,P$,CR$;
24220 GOSUB30300
24230 RETURN
24300 REM *****
24301 REM *READ RECORD FROM FILE
24309 REM *****
24310 INPUT#CH,P$
24320 P=ST
24330 GOSUB30300
24340 IS=IS+1
24350 IFF=64THENGOTO24100
24360 IFF<>0THENME$="BAD DISK STATUS"+STR$(P):GOSUB30100
24370 RETURN
24400 REM *****
24401 REM *INITIALISE DISK
24409 REM *****
24410 OPEN15,8,15,"I"+STR$(D)
24420 GOSUB30300
24430 RETURN
29800 REM *****
29801 REM *CURSOR CONTROL SUBROUTINE
29809 REM *****
29810 AC$="#####"
29820 AD$="#####"
29830 PRINT" ";
29840 IFCOL>1THENPRINTLEFT$(AC$,COL-1);
29850 IFLNE>1THENPRINTLEFT$(AD$,LNE-1);
29860 RETURN
30100 REM *****
30101 REM *WARNING MESSAGE
30109 REM *****
30110 COL=1:LNE=25:GOSUB29800
30120 A$="#####"
30130 FORA=1TO20
30140 AW=LOG( $\pi$ ):PRINT"ERROR" A$;
30150 AW=LOG( $\pi$ ):PRINT"ERROR" A$;
30160 NEXT:PRINT"ERROR  "LEFT$(ME$+SP$,32)" ";
30170 GETA$:IFA$=""THEN30170
30180 COL=1:LNE=25:GOSUB29800
30190 PRINTSP$;
30200 RETURN
30300 REM*****
30301 REM*DISK ERROR ROUTINE
30309 REM*****
30310 INPUT#15,EN,EM$,ET,ES
30320 IFEN=0GOTO30400
30330 COL=1:LNE=24:GOSUB29800
30340 PRINT"DISK ERROR - "EN;EM$;" T";ET;" S";ES;" B";DI;
30350 PRINT" CONT/ABORT ? ";
30360 GETA$:IFA$=""THEN30360
30370 IFA$<>"C"AND A$<>"A"GOTO30360
30380 IFA$="A"THEN END
30390 COL=1:LNE=24:GOSUB29800:PRINTSP$;
30400 RETURN
READY.

```

MACHINE LANGUAGE SEQUENTIAL DISK ACCESS

This section deals with a method of accessing sequential disk files from a machine code program. Of use firstly because it allows machine code programs to access disk data and secondly because subroutines written in machine code based on these ideas can be used to greatly speed up data access in Basic programs. It is this second use which will be considered in this section.

The GET# command in Basic is very slow, though often very useful, especially when concatenating data. It is useful because each character accessed from disk can be analysed to detect interfield and end of record markers, cutting out time consuming string manipulation. To overcome the problem of slow access of data using the GET# command while retaining optimum speed in concatenation it is necessary to access the disk in machine code. This is made much easier by utilising subroutines within the Basic interpreter and operating system of the PET. The following are a few of the entry points which can be used:

- \$C389 - exits to 'Ready' mode.
- \$FFCC - resets default I/O devices;
ie keyboard and screen.
- \$F770 - LDX #LF you want to access,
sets up for input.
- \$F1E1 - get character from a device,
puts char in accumulator.
- \$F7BC - LDX #LF you want to access,
sets up for output.
- \$F232 - outputs a character to a device,
accumulator contains character.
- \$E3D8 - output character to screen from acc.
- \$FFE1 - tests for the 'Stop' key and exits
to 'Break' if 'Yes'. Otherwise returns.
- \$F524 - opens file set up in LA, FA, SA.
- \$F2AE - LDA #LF that you want to close.
- \$FFE4 - get a character from keyboard buffer
character goes into acc.
- \$CA1C - prints a string terminated with a
zero and located at Y,A.
- \$C46F - input string (like Basic input), puts
string in the basic input buffer (\$0200)
- \$DB55 - floats accumulator#1. Necessary
for conversion from LO,HI to ASCII.
- \$DCE9 - Turn acc#1 into ASCII at \$0100 on.

How to convert from LO,HI to ASCII:

```
LDA #LO
STA $5F
LDA #HI
STA $60
LDX #$90
```

```
SEC
JSR $DB55
JSR $DCE9
```

ASCII number stored at \$0100 hex.

```
LA - $D2 - current logical address (for open)
FA - $D4 - current first address
SA - $D3 - current secondary address
FN - $DA,$DB - filename address pointer
FL - $D1 - file name length
```

The following machine code program demonstrates the use of these locations and entry points. It will display on the screen the contents of a sequential disk file, the name of which is input by the user in the first part of the program.

LINE#	LOC	CODE	LINE
0001	0000		*****
0002	0000		.*PROGRAM TO READ A SEQUENTIAL
0003	0000		.*FILE FROM DISK.
0004	0000		.*REQUESTS FILE NAME THEN PRINTS
0005	0000		.*CONTENTS OF FILE. DATA TRANSFER
0006	0000		.*RATE IS ABOUT 3000 BYTES PER SEC
0007	0000		*****
0008	0000		;
0009	0000		;/SYSTEM VARIABLES
0010	0000		;
0011	0000		NALO=#DA;ADDRESS OF FILE NAME LO
0012	0000		NAHI=#DB;ADDRESS OF FILE NAME HI
0013	0000		LA=#D2;LOGICAL DEVICE NUMBER
0014	0000		FA=#D4;PRIMARY ADDRESS
0015	0000		SA=#D3;SECONDARY ADDRESS
0016	0000		FL=#D1;LENGTH OF FILE NAME
0017	0000		ST=#96;STATUS CODE
0018	0000		;
0019	0000		;/SYSTEM SUBROUTINE CALLS
0020	0000		;
0021	0000		INPUT=#C46F;INPUT DATA INTO BUFFER
0022	0000		FOPEN=#F524;OPEN FILE LA,FA,SA
0023	0000		BASIN=#F1E1;INPUT SOURCE BYTE
0024	0000		PRT=#E3D8;PRINT CHARACTER TO SCREEN
0025	0000		PCLOSE=#F2AE;CLOSE FILE.A
0026	0000		CLRCHN=#FFCC;CLOSE I/O CHANNELS
0027	0000		;
0028	0000		;
0029	0000		*:=826
0030	0036		;
0031	003A	20 6F 04	START JSR INPUT
0032	003D		.*BASIC INPUT OF FILENAME INTO BUFFER
0033	003D	A9 00	LDA #00
0034	003F		.*LOAD ACC WITH LO BYTE OF FILENAME ADDRESS
0035	003F	85 DA	STA NALO
0036	0041		.*STORE IN FILENAME ADDRESS POINTER LO
0037	0041	A9 02	LDA #02
0038	0043		.*LOAD ACC WITH HI BYTE OF FILENAME ADDRESS
0039	0043	85 DB	STA NAHI
0040	0045		.*STORE IN FILENAME ADDRESS POINTER HI
0041	0045	A9 02	LDA #02
0042	0047		.*SET LOGICAL FILE TO 2
0043	0047	85 D2	STA LA
0044	0049	A9 08	LDA #08
0045	004B		.*SET DEVICE NUMBER TO 8
0046	004B	85 D4	STA FA
0047	004D	A9 02	LDA #02
0048	004F		.*SET SECONDARY ADDRESS TO 2
0049	004F	85 D3	STA SA
0050	0051	A2 00	LDX #00
0051	0053		.*SET INDEX TO ZERO
0052	0053		;
0053	0053	B0 00 02	LOOP1 LDA #0200,X
0054	0056		.*GET A CHARACTER FROM INPUT BUFFER
0055	0056	F0 04	BEQ END

LINE#	LOC	CODE	LINE
0056	0358		:IF ZERO THEN FILENAME END
0057	0358	E8	INX
0058	0359		:BUMP INDEX
0059	0359	4C 53 03	JMP LOOP1
0060	035C		:AND DO AGAIN
0061	035C		:
0062	035C	86 D1	END STX FL
0063	035E		:SET FILENAME LENGTH
0064	035E	20 24 F5	JSR FOPEN
0065	0361		:OPEN FILE
0066	0361		:
0067	0361	20 E1 F1	LOOP2 JSR BASIN
0068	0364		:GET CHARACTER FROM FILE
0069	0364	A4 96	LDY #96
0070	0366		:TEST STATUS
0071	0366	D0 06	BNE OUT
0072	0368		:END OF FILE OR ERROR DETECTED
0073	0368	20 D8 E3	JSR FRT
0074	036B		:PRINT CHARACTER
0075	036B	4C 61 03	JMP LOOP2
0076	036E		:DO AGAIN
0077	036E		:
0078	036E	A5 D2	OUT LDA LA
0079	0370		:LOAD ACC WITH LOGICAL FILE
0080	0370	20 AE F2	JSR FCLOSE
0081	0373		:CLOSE FILE
0082	0373	20 CC FF	JSR CLRCHN
0083	0376		:CLEAR CHANNELS AND RESET DEFAULT I/O
0084	0376	60	RTS
0085	0377		.END

ERRORS = 0000

SYMBOL TABLE

SYMBOL	VALUE						
BASIN	F1E1	CLRCHN	FFCC	END	035C	FA	00D4
FCLOSE	F2AE	FL	00D1	FOPEN	F524	INFUT	C46F
LA	00D2	LOOP1	0353	LOOP2	0361	NAHI	00DB
NALO	00DA	OUT	036E	PRT	E3D8	SA	00D3
ST	0096	START	033A				

END OF ASSEMBLY

RANDOM ACCESS DISK FILES

This is the most useful form of disk access, though much harder to program and more wasteful of disk space than sequential files. Whereas with a sequential file the actual position of the file on disk is unimportant, with random files we need to know the exact position (track and sector) of the data. To understand the use of random or direct (since data is accessed directly from a specified disk location) access files one must understand the way in which data is stored on disk. Data is stored in 35 concentric tracks, each track being split into between 17 and 21 sectors. Each sector or block (there are a total of 690 on a single disk) stores 255 bytes of data. Data stored in a block can be referenced by its position; block 35 is at the position of sector 3 on track 2. This is the basis of random access on the 3040 disk drive, each record is allocated a block of 255 bytes which has its own address specified by the track number and sector. Not all blocks on a disk are usable, the whole of track 18 is reserved for the disk directory and should not be used for direct access, this means that the maximum number of usable blocks on a disk is 670.

The use of direct access can be fairly complex unless certain rules are adhered to. The first rule: a whole disk must be dedicated exclusively to the storage of random data, since storing programs or sequential files on the same disk will cause problems with allocation of free blocks. The second rule: a record should not exceed 255 bytes long, otherwise problems will be encountered with block boundaries. Allied to the second rule is: when more than one record is stored in a block (this is a method of increasing the number of records on a disk) then the sum of the maximum lengths of all the records in a block should not exceed 255 bytes. The data stored in each block can be split into a number of fields, where each field is a string variable (as with sequential files, I prefer to use only one variable type in a file and thus convert all numeric variables to strings prior to writing on disk and vice versa when reading). It is very important when writing to a disk file that all blocks have the same number of fields, null data fields must be entered. Direct access records are best stored using fixed length fixed format data (see introduction). To increase the number of records above 670, each block must be divided into sub blocks. To have 1340 records means dividing each block into two 127 byte sub blocks, 2680 records four sub blocks of 63 bytes etc. Each sub record will consist of one or more fields (note: each field has a maximum length of 80 bytes, therefore a 126 byte sub-block must have a minimum of 2 fields). Data can be accessed from a direct access disk by using the track and sector number of that data to locate and read it. This method might be used in a simple stock control system where each stock line is given a number corresponding to a record number (ie the number of blocks from track 1 sector 1

to the track and sector where the data is stored) on the disk. In many cases however a data record cannot be given a number. In these cases one must use an index. The index associates a key word with the track and sector number of the data associated with that data. The index is best held as an array, in fact as two parallel arrays, one string for the keywords and the other numeric for the track and sector numbers. The key word used must be something associated with the data, thus in an address file it would be the person's name. The following is an example of how data would be held in an address file index:

Key-word	track	sector
SMITH	1	1
JONES	1	2
BROWN	1	3
WHITE	1	4

etc.

By searching through the index for a matching keyword, the track and sector where the associated data is stored can be obtained from the matching element in the numeric array. With a large key index, searching can be made easier by sorting the file into alphanumeric order (make sure that associated elements in both arrays are always kept together) and using a binary search (see machine code sort package). As data is added to the disk, the index array is updated with the new key word and the track and sector where its data is stored. The key index can be saved as a sequential file on the non direct access disk.

The subroutines associated with this section are best understood by examination and running of the example calling program.

To write a record to disk requires that a direct access data file is first opened using routine 22180. Records can then be written simply by inputting the record number DI and entering data into each of the fields in the record via an array A\$() of NF elements, where NF is the number of fields in a record. In the example, entering a 0 as record number terminates the data entry, closes the file with routine 22240, and returns to the function select menu. After entering all the records the file should be closed with routine 22240.

To read a record a direct access file is opened with routine 22180. Record number DI is then read with routine 22330, and the data placed in array A\$() from elements 1 to NF. Having read all the records required the file is closed with routine 22240 (in the example the file is closed by entering 0 as record number).

These subroutines use record numbers rather than track and sector. Subroutine 22480 -TRACK/SECTOR converts this record number into its corresponding track and sector numbers. It also stops data being written on to the directory track, track 18,(line 22525). Full error checking is done by the

subroutine DISKERROR at 30300 which uses CURSORCONT to display full error messages on line 24 and 25.

Parameters used by RANDOM:

CH - direct access channel number
D - drive number of direct access disk.
NF - number of fields in a record.
A\$()- array of NF elements in which data fields of a record are stored during record read or write.

```

100 SP$=""
110 CR$=CHR$(13)
1000 REM *****
1001 REM *DEMONSTRATION OF USE OF RANDOM
1002 REM *ACCESS FILE SUBROUTINES.
1009 REM *****
1010 CH=4
1030 INPUT"DISK DRIVE ";D
1040 INPUT"NUMBER OF FIELDS ";NF
1050 PRINT"RANDOM FILE DEMO FUNCTIONS"
1060 PRINT"1-WRITE RECORD"
1070 PRINT"2-READ RECORD"
1080 PRINT"3-END"
1090 GETA$:IFA$="" THEN 1090
1100 A=VAL(A$)
1110 ONA GOTO 1200,1500,1120
1120 END
1200 REM **WRITE RECORD TO DISK**
1210 GOSUB 22180
1220 PRINT"WRITE RECORD TO DISK"
1230 INPUT"RECORD NUMBER ? ";DI
1240 IF DI=0 THEN GOSUB 22240:GOTO 1050
1250 PRINT""
1260 FOR Q=1 TO NF
1270 PRINT"FIELD ";Q;" IS - ";
1280 INPUT A$(Q)
1290 NEXT
1300 GOSUB 22080
1310 GOTO 1220
1500 REM **READ RECORD FROM DISK**
1510 GOSUB 22180
1520 PRINT"READ RECORD FROM DISK"
1530 INPUT"RECORD NUMBER ? ";DI
1540 IF DI=0 THEN GOSUB 22240:GOTO 1050
1550 PRINT""
1560 GOSUB 22330
1570 FOR Q=1 TO NF
1580 PRINT"FIELD ";Q;" IS - ";A$(Q)
1590 NEXT Q
1600 GET A$:IFA$="" THEN 1600
1610 GOTO 1520
2000 REM
3000 REM
4000 REM
5000 REM
22080 REM *****
22090 REM *ROUTINE TO WRITE RANDOM DATA
22100 REM *****
22110 GOSUB 22480
22120 PRINT#15,"U1"CH;D;DT;DS
22130 PRINT#15,"B-F"CH,1
22140 FOR Q=1 TO NF
22150 PRINT#CH,A$(Q);CR$;
22160 NEXT
22170 PRINT#15,"U2"CH;D;DT;DS
22172 GOSUB 30300
22175 RETURN
22180 REM *****

```


DISK UTILITIES

The two programs in this section are designed to give the user information about how data is being stored on disk; they are of most use when using direct access files. BLOCKMAP prints a table of all the blocks on a disk, track by track, showing which blocks have been allocated for data storage by the system and which are free. Free blocks can then be used for direct access (modifying the subroutine TRACK/SECTOR one can block off allocated sectors). Alternatively one may simply wish to find out how much free space is left on a disk. SECTORPRINT is the second of the two programs and is designed to print a table of data stored in a specified block. The data is displayed both in ASCII and hexadecimal form. Using this program one can see how data is stored on disk and also discover the source of disk errors or corruption.

```

100 REM *****
110 REM *THIS DISK UTILITY PROGRAM
120 REM *ALLOWS ONE TO LOOK AT WHICH
130 REM *BLOCKS ON THE DISK ARE
140 REM *ALLOCATED TO DATA OR PROGRAMS.
150 REM *USEFUL IN CHECKING WHERE DATA
160 REM *HAS BEEN STORED WHEN USING
170 REM *RANDOM FILES.
180 REM *****
1000 POKE144,49: REM *DISABLE STOP KEY*
1010 OPEN15,8,15
1020 PRINT"*****"
1030 PRINT"HARDCOPY Y OR N ":P=0
1040 GETA$:IFA$=""THEN1040
1050 IFA$="Y"THENP=1
1060 PRINT"DISK DRIVE# "
1070 GETD$:IFD$=""THEN1070
1080 D=VAL(D$)
1090 IFFTHENOPEN1,4:PRINT#1:PRINT#1
1100 PRINT"D"
1110 P1$="12"
1120 P2$="TRACK 012345678901234567890 USED FREE"
1130 PRINTP1$:PRINTP2$
1140 IFFTHENPRINT#1,P1$:PRINT#1,P2$
1150 FORDT=1T035
1160 DT$=RIGHT$(STR$(100+DT),2)+" "
1170 ADS=21
1180 IFDT>17THENADS=20
1190 IFDT>24THENADS=18
1200 IFDT>30THENADS=17
1210 PRINTDT$:IFFTHENPRINT#1,DT$;
1220 AU=0:AF=0
1230 FORDS=0T020
1240 A$="."
1250 IFDS>=ADSTHENA$=" ":GOTO1320
1260 PRINT#15,"B-A";D;DT;DS
1270 INPUT#15,EN,E1$,E2$,E3$
1280 IFEN<>0THENA$="*":AU=AU+1:GOTO1320
1290 AF=AF+1:PRINT#15,"B-F";D;DT;DS
1300 GOSUB1440:IFEN=0THEN1320
1310 PRINT:PRINT:PRINT"DISK ERROR"EN,E1$,E2$,E3$:STOP
1320 PRINTA$:IFFTHENPRINT#1,A$;
1330 NEXTDS
1340 AU$=RIGHT$(STR$(AU+100),2)+" "
1350 AU$=AU$+RIGHT$(STR$(AF+100),2)
1360 PRINT" "AU$:IFFTHENPRINT#1," ";AU$
1370 BU=BU+AU:BF=BF+AF
1380 NEXTDT
1390 PRINT:PRINT
1400 P3$="BLOCKS USED "+STR$(BU)+" FREE "+STR$(BF)
1410 PRINTP3$:IFFTHENPRINT#1,P3$
1420 CLOSE1:CLOSE15
1430 POKE144,46:END
1440 INPUT#15,EN,E1$,E2$,E3$
1450 RETURN
READY.

```

	1	2		
TRACK	012345678901234567890	USED	FREE	
01	00	21	
02	00	21	
03	00	21	
04	00	21	
05	00	21	
06	00	21	
07	00	21	
08	00	21	
09	00	21	
10	00	21	
11	00	21	
12	*****	21	00	
13	*****	21	00	
14	*****	21	00	
15	*****	21	00	
16	*****	21	00	
17	*****	21	00	
18	**..*..*..*..*..*	08	12	
19	*****	20	00	
20	*****	20	00	
21	*****	20	00	
22	*****	20	00	
23	*****	20	00	
24	*****	20	00	
25*.....	01	17	
26	00	18	
27	00	18	
28	00	18	
29	00	18	
30	00	18	
31	00	17	
32	00	17	
33	00	17	
34	00	17	
35	00	17	
BLOCKS USED	255	FREE	435	

Sample printout from program BLOCKMAP

```

100 REM *****
110 REM *DISK UTILITY PROGRAM TO PRINT
120 REM *CONTENTS OF A DISK SECTOR IN
130 REM *HEX AND ASCII. USEFUL IN CHECKING
140 REM *DATA STORED CORRECTLY.
150 REM *****
1000 DIMA(264)
1010 Z1$=CHR$(0)
1015 PRINT"J"
1020 PRINT"HARDCOPY Y OR N ";:P=0
1025 GETA$:IFA$=""THEN1025
1030 IFA$="Y"THENP=1
1040 OPEN4,8,4,"#":CH=4
1050 OPEN15,8,15
1060 IFPTHENOPEN1,4:PRINT#1,CHR$(147);
1070 PRINT"J"
1080 INPUT"DRIVE ";D
1090 INPUT"TRACK ";T
1100 INPUT"SECTOR ";S
1110 PRINT"XXXX"
1120 PRINT#15,"U1"CH;D;T;S:GOSUB1490
1130 PRINT#15,"B-P"CH;1
1140 FORI=1TO250STEP5:GET#4,C1$,C2$,C3$,C4$,C5$
1150 A(I)=ASC(C1$+Z1$)
1160 A(I+1)=ASC(C2$+Z1$)
1170 A(I+2)=ASC(C3$+Z1$)
1180 A(I+3)=ASC(C4$+Z1$)
1190 A(I+4)=ASC(C5$+Z1$)
1200 NEXTI
1210 FORI=1TO250STEP8
1220 IFI=121THENGOSUB1460
1230 A$=RIGHT$(STR$(1000+I),3)+": "
1240 PRINTA$;:IFPTHENPRINT#1,A$;
1250 FORQ=0TO7
1260 IFA(I+Q)=13THENA$="+":GOTO1300
1270 IFA(I+Q)=10THENA$="<":GOTO1300
1280 IFA(I+Q)>127ORA(I+Q)<32THENA$="." :GOTO1300
1290 A$=CHR$(A(I+Q))
1300 PRINTA$;:IFPTHENPRINT#1,A$;
1310 NEXTQ
1320 PRINT" ";:IFPTHENPRINT#1," ";
1330 FORQ=0TO7
1340 A=A(I+Q)
1350 A1=ARAND15:A0=(ARAND240)/16
1360 A1=A1+48:IFA1>57THENA1=A1+7
1370 A0=A0+48:IFA0>57THENA0=A0+7
1380 PRINT" ";CHR$(A0);CHR$(A1);
1390 IFPTHENPRINT#1," ";CHR$(A0);CHR$(A1);
1400 NEXTQ:PRINT" ";:IFPTHENPRINT#1," "
1410 NEXTI
1420 PRINT"XX":IFPTHENPRINT#1
1430 PRINT"ANOTHER -YES OR NO-";
1435 GETA$:IFA$=""THEN1435
1440 IFA$="Y"THEN1070
1450 CLOSE4:CLOSE15:END
1460 GETA$:IFA$=""THEN1460
1470 PRINT"J"
1480 RETURN
1490 INPUT#15,EN,E1$,E2$,E3$
1495 IFEN>0THENSTOP
1500 RETURN
READY.

```

001:.....BLOC	04	82	11	00	42	4C	4F	43
009:KMAP.....	4B	4D	41	50	A0	A0	A0	A0
017:.....	A0	A0	A0	A0	00	00	00	00
025:.....	00	00	00	00	00	04	00	00
033:.....SECT	00	82	11	01	53	45	43	54
041:ORPRINT.	4F	52	50	52	49	4E	54	A0
049:.....	A0	A0	A0	A0	00	00	00	00
057:.....	00	00	00	00	00	05	00	00
065:.....DIGI	00	82	11	04	44	49	47	49
073:T.....	54	A0	A0	A0	A0	A0	A0	A0
081:.....	A0	A0	A0	A0	00	00	00	00
089:.....	00	00	00	00	00	02	00	00
097:.....RAND	00	82	11	05	52	41	4E	44
105:OM.....	4F	4D	A0	A0	A0	A0	A0	A0
113:.....	A0	A0	A0	A0	00	00	00	00
121:.....	00	00	00	00	00	05	00	00
129:.....INIT	00	82	11	09	49	4E	49	54
137:RAND.....	52	41	4E	44	A0	A0	A0	A0
145:.....	A0	A0	A0	A0	00	00	00	00
153:.....	00	00	00	00	00	06	00	00
161:.....BDIS	00	82	13	00	42	44	49	53
169:F.....	50	A0	A0	A0	A0	A0	A0	A0
177:.....	A0	A0	A0	A0	00	00	00	00
185:.....	00	00	00	13	00	04	00	00
193:.....DDPL	00	82	13	05	44	44	50	4C
201:OT.....	4F	54	A0	A0	A0	A0	A0	A0
209:.....	A0	A0	A0	A0	00	00	00	00
217:.....	00	00	00	13	05	06	00	00
225:.....RAND	00	82	13	02	52	41	4E	44
233:DEMO.....	44	45	4D	4F	A0	A0	A0	A0
241:.....	A0	A0	A0	A0	00	00	00	00
249:.....	00	00	00	00	00	00	00	00

TRACK 18 SECTOR 1

Sample printout from program SECTORPRINT

LINKING PROGRAMS AND SUBROUTINES WITH MENUS

A menu is simply a program or subroutine which displays the different options available to the user in a program or program suite. The user can then use the menu to select an option, and the menu routine will either load the program or cause the program to jump to the start of the specified subroutine. A menu subroutine within a program is very simple, consisting of code to print the options list and a computed 'GOTO'. Examples of this type of routine are included in several programs in this book (eg: machine code sort package). A menu program to link and select different programs in a program suite is more complex, the following program (MENU) is an example. In this program lines 10 to 150 print the option menu on the screen. Since this is an example the options are just called Program 1, Program 2 etc, in a real application this would be replaced by a description of the function. Lines 160 to 220 perform the function of loading the required program from disk drive zero. Line 160 inputs a number which is equivalent to one of the numbers preceding the option description. In MENU this can be a number between 1 and 6. The selected program is then loaded by one of the lines between 170 and 220. A special trick is required to load one program from another using the disk, the trick requires the manipulation of the pointers to the start of the variable storage area in memory. These pointers are contained in locations 42 and 43, and should be loaded with the start of variable storage location for the program which is to be loaded. These values can be obtained by loading each program in the suite and PEEKing locations 42 and 43, noting the values for each program. In the example menu program the POKE values for locations 42 and 43 for each program are set to 0 since there are obviously no programs, the correct values should be inserted. The reason each POKE value has three digits is that the values of locations 42 and 43 for the menu program must also be known before any values are inserted. If all numbers are entered as three digits, with leading digits zeros if required, then the menu program length will remain constant and the values for 42 and 43 will be constant. The values of the start of variables pointers for the menu are needed, since we want the menu program to be automatically loaded after each program in the suite has ended, instead of using the END command. See MERGE line 1308, page 82, where sample values for locations 42 and 43 are given. These should be changed as necessary.

```

1 REM *****
2 REM *MENU PROGRAM TO LINK DIFFERENT
3 REM *PROGRAMS IN A SUIT OF PROGRAMS
4 REM *TOGETHER.
9 REM *****
10 PRINT "  MENU"
20 PRINT "  ----"
30 PRINT
40 PRINT "    1    PROGRAM 1"
50 PRINT
60 PRINT "    2    PROGRAM 2"
70 PRINT
80 PRINT "    3    PROGRAM 3"
90 PRINT
100 PRINT "    4    PROGRAM 4"
110 PRINT
120 PRINT "    5    PROGRAM 5"
130 PRINT
140 PRINT "    6    END"
150 PRINT
160 GETA$: IFA$="" THEN 160
170 IFA$="1" THEN POKE 42,000: POKE 43,000: CLR: LOAD "0:PROGRAM 1",8
180 IFA$="2" THEN POKE 42,000: POKE 43,000: CLR: LOAD "0:PROGRAM 2",8
190 IFA$="3" THEN POKE 42,000: POKE 43,000: CLR: LOAD "0:PROGRAM 3",8
200 IFA$="4" THEN POKE 42,000: POKE 43,000: CLR: LOAD "0:PROGRAM 4",8
210 IFA$="5" THEN POKE 42,000: POKE 43,000: CLR: LOAD "0:PROGRAM 5",8
220 IFA$=>"6" THEN END
READY.

```

MISCELLANEOUS ROUTINES

32TRACE

This machine code utility program is intended for use with machines incorporating Basic 3.00 ROMs. Its function is to display each line of a Basic program as it is executed thus enabling the programmer to follow the logic flow of the program and detect where errors are occurring. The Basic loader for TRACE is loaded and run then initialised using the SYS command given by the loader program, prior to loading the Basic program to be tested. Having loaded the program, TRACE should be enabled using the SYS command given in the loader and the program run by typing RUN in the normal manner. Each line of the program will then be displayed in reverse field on line one of the screen as it is executed.

DATAMAKER

This program is designed to create a series of data statements containing the values of successive memory locations containing a machine code program. The data statements created can then be used with a simple Basic loader to load the values back into memory from a Basic program. DATAMAKER uses a technique of getting the computer to write its own program lines (see 'THE PET REVEALED' for further information on this subject). Having created the data statements, DATAMAKER should be deleted (to aid this it has been given line numbers from 60000 to 62000).

32REPEAT

This is another machine code program which allows a repeat key function to be added to a 16K or 32K Basic 3.00 PET. The Basic loader is loaded when the machine is first switched on and the repeat key function will remain in the machine until it is switched off or the second cassette buffer is used. Once the repeat program is loaded the repeat function can be enabled with the command SYS(832), then any key will repeat if held down long enough.

AUTOL

A Basic program to automatically number lines when writing a program. Given the starting line number and line number increment, each line number will be displayed on the screen as lines are written. Simply enter the program line after the number and press return. The line will be entered into the program and the next line number displayed. Press the stop

key to stop the program. By running the program and entering a line number and increment of program lines to be deleted then pressing return after each line number this routine can be used to delete lines.

SCREENPRINT

This little subroutine will transfer any alphanumeric data on the screen on to the printer in the same format as displayed on screen. This routine is intended for the 3022 printer and variable PN controls the interline spacing on the printer. It should be adjusted to suit the application.

```

1 REM *****
2 REM *TRACE PROGRAM FOR BASIC 3.00 ROM
3 REM *MACHINES. DISPLAYS EACH LINE OF A
4 REM *PROGRAM ON THE SCREEN AS IT IS
5 REM *EXECUTED. USE SYS COMMANDS SHOWN
6 REM *WHEN RUNNING THE PROGRAM. NOTE
7 REM *THESE VALUES EACH TIME SINCE THEY
8 REM *WILL VARY.
9 REM *****
10 E=52
15 D=2
20 DATA-342,162,5,189,249,224,149,112,202,16,248,169,239,133,128,96
30 DATA173,-342,133,52,173,-341,133,53,169,255,133,42,160,0,162,3
40 DATA134,43,162,3,32,-271,208,249,202,208,248,32,-271,32,-271,76
50 DATA121,197,162,5,189,-6,149,112,202,16,248,169,242,133,128,96
60 DATA230,42,208,2,230,43,177,42,96,230,119,208,2,230,120,96
70 DATA32,115,0,8,72,133,195,138,72,152,72,166,55,165,54,197
80 DATA253,208,4,228,254,240,106,133,253,133,35,134,254,134,36,165
90 DATA152,208,14,169,3,133,107,202,208,253,136,208,250,198,107,208
100 DATA246,32,-54,169,160,160,80,153,255,127,136,208,250,132,182,132
110 DATA37,132,38,132,39,120,248,160,15,6,35,38,36,162,253,181
120 DATA40,117,40,149,40,232,48,247,136,16,238,216,88,162,2,169
130 DATA48,133,103,134,102,181,37,72,74,74,74,32,-44,104,41
140 DATA15,32,-44,166,102,202,16,233,32,-38,32,-38,165,184,197,119
150 DATA240,55,165,195,208,4,133,253,240,47,16,42,201,255,208,8
160 DATA169,105,32,-30,24,114,33,41,127,170,160,0,135,145,192,48
170 DATA3,200,208,248,200,202,16,244,185,145,192,48,6,32,-32,200
180 DATA208,245,41,127,32,-32,165,119,133,184,104,168,104,170,104,40
190 DATA96,168,173,64,232,41,32,208,249,152,96,9,48,197,103,208
200 DATA4,169,32,208,2,198,103,41,63,9,128,132,106,32,-54,164,182
210 DATA153,0,128,192,195,208,2,160,7,200,132,182,164,106,96,76
220 DATA-255,32,-262
300 S2=PEEK(E)+PEEK(E+1)*256:S1=S2+D-344
310 FORJ=S1TO S2-1
320 READX:IFX=0ORX=0THENGOTO350
330 Y=X+S2:X=INT(Y/256):Z=Y-X*256
340 POKEJ,Z:J=J+1
350 POKEJ,X
360 NEXTJ
400 PRINT"*****32TRACE"
410 PRINT"*****"
500 PRINT"INITIALISE WITH SYS(;"S1+17";)"
510 PRINT"ENABLE WITH SYS(;"S1+56";)"
520 PRINT"DISABLE WITH SYS(;"S1+2";)"
530 PRINT"CHANGE SPEED WITH POKE;"S1+125-D";,X"
540 END
READY.

```

```

1000 REM *****
1001 REM *THIS PROGRAM WILL CONVERT A
1002 REM *MACHINE CODE PROGRAM STORED IN
1003 REM *MEMORY INTO A SERIES OF DATA
1004 REM *STATEMENTS WHICH CAN BE USED
1005 REM *BY A BASIC LOADER PROGRAM TO
1006 REM *LOAD THE MACHINE CODE INTO THE
1007 REM *FROM BASIC. SPECIFY THE START
1008 REM *AND END MEMORY LOCATION OF THE
1009 REM *MACHINE CODE (IN DECIMAL).
1010 REM *THE DATA STATEMENTS CONTAIN THE
1011 REM *VALUES OF EACH BYTE IN DECIMAL.
1012 REM *ALSO SPECIFY THE STARTING LINE
1013 REM *NUMBER AND INCREMENT OF THE
1014 REM *DATA STATEMENTS (LINE 60000).
1020 REM *****
60000 INPUT "START#,STEP";S,T
60010 INPUT "START ADDRESS DECIMAL";B
60020 F=B:L=F+10
60030 INPUT "END ADDRESS DECIMAL";E
60050 PRINT "*****"
60060 POKE831,INT(E/256)
60070 POKE832,E-INT(E/256)*256
60100 POKE828,T:GOTO60500
60200 S=PEEK(826)*256+PEEK(827)
60300 T=PEEK(828)
60310 L=PEEK(829)*256+PEEK(830)
60330 E=PEEK(831)*256+PEEK(832)
60340 IFL>=EGOTO62000
60350 F=L+1:L=L+10
60400 PRINT";"
60500 PRINTS;
60600 PRINT"DATA";
60700 FORP=FTOL:PRINTPEEK(P);" ";:NEXTP
60800 PRINT""
60900 PRINT"GOTO60200"
61000 POKE158,2:POKE623,13:POKE624,13
61100 S=S+T
61200 POKE826,INT(S/256)
61300 POKE827,S-INT(S/256)*256
61400 POKE829,INT(L/256)
61500 POKE830,L-INT(L/256)*256:END
62000 STOP
READY.

```

"32REPEAT"

```
1 REM *****
2 REM *BASIC LOADER FOR
3 REM *MACHINE CODE PROGRAM TO ADD A
4 REM *REPEAT KEY FUNCTION TO A 32 K
5 REM *PET. HOLDING THE KEY DOWN WILL
6 REM *CAUSE IT TO REPEAT. INITIALISE
7 REM *WITH SYS(832).
9 REM *****
10 FORQ=832TO891 0340-037B
20 READA
30 POKEQ,A
40 NEXTQ
50 END
100 DATA120,169,79,133,144,169,3,133,145,169
110 DATA1,133,2,88,96,165,151,197,0,240,9
120 DATA133,0,169,16,133,1,76,46,230,201,255
130 DATA240,249,165,1,240,4,198,1,208,241
140 DATA198,2,208,237,169,4,133,2,169,0,133
150 DATA151,169,2,133,168,208,223
READY.
```

"AUTOL"

```
1000 REM *****
1010 REM *ROUTINE TO AUTOMATICALLY
1020 REM *NUMBER LINES WHEN WRITING
1030 REM *A PROGRAM. ENTER START
1040 REM *LINE AND LINE INCREMENT
1050 REM *THEN ENTER PROGRAM LINES
1060 REM *AFTER LINE NUMBER.
1070 REM *****
60000 INPUT"START#,STEP";S,T
60050 PRINT"*****"
60100 POKE828,T:GOTO60500
60200 S=PEEK(826)*256+PEEK(827)
60300 T=PEEK(828)
60400 PRINT" "
60500 PRINTS;
60700 GETD$:IFD$=""THEN60700
60800 PRINTD$;:IFASC(D$)>13THEN60700
60900 PRINT"GOTO60200TT";
61000 POKE158,2:POKE623,13:POKE624,13
61100 S=S+T
61200 POKE826,INT(S/256)
61300 POKE827,S-INT(S/256)*256:END
READY.
```

```
10 FN=24
20 SP$=""
25000 REM *****
25001 REM *SCREEN PRINTER SUBROUTINE
25009 REM *****
25010 OPEN3,4,6:PRINT#3,CHR$(FN)
25020 OPEN4,4:PRINT#4,"":PRINT#4,SP$;
25030 FORQ=0TO999:A=PEEK(Q+32768)
25040 B=(AAND127)OR((AAND64)*2)OR((64-AAND32)*2)
25050 PRINT#4,CHR$(B);
25060 X=X+1:IFX=40THENPRINT#4,"":PRINT#4,SP$;:X=0
25070 NEXT:CLOSE4
25080 RETURN
READY.
```